UNIVERSITY OF CAMBRIDGE

A Theory of Dynamic Software Updates

A thesis submitted for the Degree of Doctor in Philosophy,
by Gareth Paul Stoyle of Hughes Hall.

To my parents

Abstract

This thesis addresses the problem of evolving software through a sequence of releases without halting execution, a process referred to as *Dynamic Software Updating* (DSU). It looks at the theoretical foundations, develops an applied theory, and shows how this can be used to automatically transform programs into upgradable ones that come with guarantees of updatability. In contrast to many previous approaches, our semantics are developed at the language level, allowing for on-line evolution to match source-code evolution.

The first part of the thesis takes a foundational approach, developing a core theory of dynamic rebinding. The theory looks afresh at the reduction semantics of the call-by-value (CBV) λ-calculus, delaying instantiations so that computations always use the most recently rebound version of a definition. We introduce the *redex-time* and *destruct-time* strategies that differ in how long they delay instantiations. The computational consistency of these calculi are confirmed by showing that their contextual equivalence relations agree with that of classical CBV.

The second part of the thesis presents Proteus, a core calculus for dynamic software updating in C-like languages that is flexible, safe, and predictable. Proteus supports dynamic updates to functions (even active ones), to named types, and to data. First it is shown how an *a posteriori* notion of abstraction can lead to a very general type-directed mechanism for DSU whose safety can be assured dynamically, and second that this dynamic check has a good static approximation. The latter is shown by constructing a novel capability-based type system that is proved sound with respect to a reduction semantics. The final chapter reports on a prototype implementation of the theory for the C programming language.

Declaration

This thesis:

- is my own work and contains nothing which is the outcome of work done in collaboration with others except as specified in the text;
- is not substantially the same as any I have submitted for a degree or diploma or other qualification at any other university; and
- does not exceed the prescribed limit of 60,000 words.

Gareth Stoyle
Cambridge, England, 2006

Acknowledgements

Firstly, I would like to thank my research supervisors Gavin Bierman and Peter Sewell who have provided invaluable knowledge, teaching and guidance throughout my PhD studies. I have learnt much from them and to both of them, I am very grateful. I also owe thanks to my friends and colleagues at the Computer Laboratory for acting in many different capacities. Tea times provided much needed distraction and would not have been such fun without Matthew Parkinson, Lucy Saunders-Evans, Andrei Serjantov, Mark Shinwell, Sam Staton, Daniele Varacca and Alisdair Wren, many of whom also provided me with invaluable advice and help whenever I was stuck.

During my studies I have had the great pleasure of working with Mike Hicks and Iulian Neamtiu. Mike Hicks deserves a special thanks for sharing with me his expert knowledge in the area of dynamic software updating and for the many fruitful and enjoyable discussions we had about the subject. I would also like to thank Mike, his family and Iulian Neamtiu for their generous hospitality during my two month stay in Maryland.

I must thank Graham Birtwistle, as without his encouragement I may never have applied for a PhD at Cambridge.

Finally, I should like to thank my family for the constant support and encouragement they have shown in all that I do. Without them this thesis would not have been possible. Also, I would never have found my way through the sometimes arduous task of writing up without the love and support of Jessica Marples.

Contents

Contents **vii**

1 Introduction **1**
1.1 Approaches to DSU 2
1.1.1 A systems approach 2
1.1.2 A software approach 2
1.2 Understanding Dynamic Relinking 4
1.2.1 CBV Equivalence 7
1.2.2 Safe Dynamic Relinking 7
1.3 Dynamic Update 8
1.3.1 After the Event Abstraction 10
1.3.2 A Capability Approach 11
1.4 Pragmatic Update 13
1.5 Update Systems Introduced in This Thesis 13
1.5.1 Lambda-Update 13
1.5.2 Update Calculus 14
1.5.3 Proteus 14
1.5.4 Proteus$^{\Delta}$ 14
1.6 Related Work 14
1.6.1 λ and Delayed Instantiation 15
1.6.2 Dynamic Update 15
1.6.3 Capability Type Systems 20
1.7 Collaboration 20

I The λ-calculus and updatability **21**

2 Delayed instantiation **23**

2.1 The failure of CBV semantics 24
2.2 CBV λ-calculus revisited 25
2.2.1 Construct-time 26
2.2.2 Redex-time 26
2.2.3 Destruct-time 30
2.2.4 Properties 32
2.3 Update calculus 33
2.3.1 Update mechanism 34
2.3.2 Conclusions 38

3 λ_c,λ_r,λ_d Equivalence **39**
3.1 Equivalence 39
3.2 Observational Equivalence Between λ_r and λ_c 41
3.2.1 An Overview 43
3.2.2 An Eventually Weak Bisimulation Relation 44
3.2.3 R is an Eventually Weak Bisimulation 59
3.2.4 Equivalence 81
3.3 Observational Equivalence Between λ_d and λ_c 86
3.3.1 Transforming proofs from λ_r to λ_d 88
3.3.2 Changed Proofs 89
3.4 Conclusion 100

4 Update Via Versioned Modules **101**
4.1 The update calculus 102
4.1.1 Syntax 102
4.1.2 Semantics 104
4.1.3 Discussion 107
4.2 Update Strategies 108
4.2.1 Stop and Switch 108
4.2.2 Complete and Switch 109
4.2.3 Convert and Switch 110
4.3 Conclusions 111

II Proteus **113**

5 Proteus **115**

5.1 Proteus Source Language 116
5.1.1 Example 118
5.1.2 Update 119
5.1.3 Proteus Target Language 120
5.2 Automatic coercion insertion 121
5.3 Specifying Dynamic Updates 130
5.4 Operational Semantics 134
5.4.1 Replacing Top-level Identifiers 137
5.4.2 Updating Data of Named Type 139
5.4.3 Update Semantics 139
5.5 Update Safety 139
5.5.1 Update Timing 141
5.5.2 Update well-formedness 142
5.6 Properties 142
5.7 Conclusion 144

6 Update Capability Analysis **145**
6.1 Capabilities 145
6.2 Typing 146
6.2.1 Expression Typing 149
6.2.2 Program Typing 152
6.2.3 Configuration Typing 153
6.3 Operational Semantics 153
6.3.1 Phase Order 156
6.4 Inference 156
6.4.1 Inferring update Points 158
6.5 Properties 158
6.6 Extensions 159
6.6.1 Binding Deletion 159
6.6.2 Recursive types 160
6.7 Conclusions 160

7 Implementation **161**
7.1 Writing Updatable Programs 161
7.1.1 An example 163
7.1.2 Tools 167

7.2 Program compilation . . . 168
7.2.1 Coercion insertion . . . 169
7.2.2 Update Analysis . . . 170
7.2.3 Layout . . . 171
7.2.4 Version Data . . . 174
7.3 Abstraction-Violating Alias Analysis . . . 174
7.3.1 Description . . . 175
7.4 Patch compilation . . . 176
7.4.1 Patch Generation . . . 177
7.5 Performance . . . 177
7.6 Patterns for Updatable Programs . . . 178
7.7 Conclusions . . . 179

8 Conclusions and Future Work 181
8.1 Summary . . . 181
8.2 Future Work . . . 182
8.2.1 Integration with Other Language Features . . . 182
8.2.2 Concurrency . . . 182
8.2.3 Implementation . . . 183

Appendices 189

A Proof of Type Soundness for Proteus 191

B Proof of Type Soundness for Proteus-Delta 197
B.1 Proof of Type Soundness . . . 198

Bibliography 221

1 Introduction

Software changes. It changes because the requirements change, because a bug is found, or because an optimisation is discovered. Whatever the reason, change, in the form of software upgrades, pervades the life cycle of any non-trivial software product.

The usual approach to a software upgrade involves the developer providing a new executable, the client halting execution of the existing system and restarting with the new. In the case of critical services such as financial transaction processors, air traffic control and network infrastructure a *planned outage* is organised for the upgrade, possibly resulting in substantial financial loss. Dynamic Software Updating (DSU) is an approach that avoids this temporary interruption caused by a software upgrade by allowing the system to be *patched* on-the-fly.

Of course, the techniques of DSU are not limited to critical systems: for other non-critical but continuous services, such as the SSH daemon or a personal computer's operating system kernel, it is not necessary, but nonetheless convenient to upgrade them without interruption. An important instance of this occurs when software is managed asynchronously to its use, such as a service centre changing the software on a mobile phone which may be simultaneously in use by the client. In these situations DSU can allow the change to happen without the need to notify the user.

DSU also has applications that are not related to software upgrades. Examples of this are debugging and performance tuning. DSU allows the engineer to augment running software to collect information about problems that occur with a production system, such as poor performance or intermittent faulty behaviour. The information required to diagnose and fix the problem is not known in advance, and certainly not at compile time, so when a fault is noticed a software extension can be installed to determine the cause.

This thesis investigates the theory of updatable systems and how programs can be automatically compiled to allow updates at runtime.

1.1 Approaches to DSU

The purpose of DSU is to dynamically change the behaviour of a running software system without interruption. We group approaches to the problem under the broad headings of systems-based and software-based approaches.

1.1.1 A systems approach

As a first thought we may decide the simplest way to achieve DSU is through the use of redundant hardware. The idea here is that two machines are available, A and B. A runs the application until an upgrade is required, at which point B is brought up running the new software, the state of the application (current connections, open files, the state of any computations, etc.) is transferred from A to B, and B takes over. The main problem to be solved is how to transfer the state. More precisely, how is the relevant state extracted from A, transformed to be compatible with the new software and injected into B? How is the system dependent state such as open sockets and file handles transferred from A to B? And when is a safe time for this to be performed?

Taking a redundant hardware approach ties together the dynamic upgrade with the complications of process migration, while still retaining the problem of transferring the state between the old and new version. If no state transfer is needed, such as in the case of a server with a stateless protocol (e.g. HTTP) then this approach is a fine solution. However, for the majority of cases where state transfer is required we would be better upgrading a single address space and not introducing the extra problem of migration between machines.

1.1.2 A software approach

Having observed that redundant hardware does not help us we shall concentrate directly on a software approach, focusing on the problem of transforming the state of the old process into a corresponding state for the new process, along with installing the new code and data. The central challenge is to discover mechanisms that allow us to structure dynamic changes to a system.

If we accept that writing correct software is a hard task in its own right, then writing a *patch* to a running system is very hard, as when extending a software system dynamically we not only have to think about logical correctness, but also temporal correctness. This temporal correctness refers to the fact that not all changes will only be valid at all execution points: we probably do not want to change an integer to a string just before addition is about to take place.

Dynamic update can be seen as consisting of three parts: dynamic linking, re-linking and state transfer.

Dynamic linking This is a well understood stage where new code and data are made accessible to the program at runtime. A prototypical implementation is POSIX's dlopen[1] which can load data and code from shared libraries into a process' address space and provides a mechanism to locate items in the library by name.

Re-linking After new definitions have been loaded, it is necessary to ensure not only that new code uses the new definitions, but also that old code does too. Existing binders in the old code are *re-linked* to their new definitions, which may have changed type.

State transfer When a program is upgraded the data structures it uses will most likely change. For example, a data structure that was previously implemented as a singly-linked list may now be required to be doubly-linked, in which case every instance of the list should have reverse links added. Another case might require storage expansion, where locations previously assumed to be integers are upgraded to records, or indeed vice-versa where storage is contracted, the record being replaced with an index to locate it.

During all three stages we would like to maintain some notion of *safety*. Here we are interested in ensuring that the program will not crash by maintaining type safety. As mentioned before, with a patch we not only have to check that it is safe to apply, but also that it is safe to apply with respect to a program state, or set of program states.

The first part of this thesis is concerned with re-linking at the fundamental level of the λ-calculus, while the latter half examines how a form of *a posteriori* abstraction can be used to ensure representation consistency during state transfer.

[1]See the UNIX man page `dlopen(3)`

1.2 Understanding Dynamic Relinking

The linking of an identifier to a value traditionally occurs at either compile time, link time or runtime. Compile time resolution of an identifier to the value it binds usually applies to local variables, link time binding to occurrences of identifiers belonging to one module in another and runtime linking to identifiers that are shared between many programs, such as interfaces to system resources.

Our motivation for looking at linking is in the context of dynamic updates, where we are interested not just in linking, but *relinking*. A minimal requirement of dynamic update is to be able to change a program at runtime by relinking its identifiers to new code and data. While dynamic linking is well understood [DE02, Car97], there is a much more patchy understanding of the dynamic relinking required for DSU.

A simple approach to relinking at runtime, allowing limited dynamic changes, are plugin architectures. A program that has a plugin architecture delays choices about the number and type of particular services it provides until runtime. It exports an interface for a service (such as a filter facility in a graphics package) and looks for shared libraries providing this interface dynamically at runtime. Plugin architectures are an explicit realization of dynamic rebinding at runtime. The programmer designs his program to allow a predetermined set of variables, referring to the given service, to be rebound at runtime. This limited form of update has proved successful in contexts where third parties are expected to extend software in a specific direction, such as adding extra filters to graphics packages.

The plugin approach does not address unexpected changes. For this we want to make *changes* to the original program, not just extensions. The most natural way to make changes to a program is by adding new variables and rebinding existing variables to new values. Thus, dynamic changes must be understood in the context of the original program and we are led to consider dynamic relinking in the context of programming language semantics.

Despite considerable previous interest in DSU (see related work in §1.6), there has, perhaps surprisingly, been little formalism. Therefore, a suitable place to begin is by considering how dynamic update can be understood in a traditional model of a programming language, by which we mean the λ-calculus with a let construct, paring and projection (written π_i). We assume the reader to be familiar with such a system.

The let construct allows terms to be labelled with an identifier, providing a natural unit of *indirection*.

$$\textbf{let}\ z = (5,4)\ \textbf{in}\ z$$

While the above example program refers to z, it is not committed to the actual value presently bound to z. In other words, the program has the potential to notice any change in the binding of z. After the occurrences of z are replaced by its definition the indirection is collapsed and a choice is made to use the current definition. There are several choices that can be made about how to collapse this indirection, i.e. how to instantiate variables. The usual one is substitution.

Recall the usual call-by-value (CBV) evaluation strategy for the λ-calculus and consider the following reduction:

$$\textbf{let}\ z = (5,4)\ \textbf{in}\ \pi_1(\pi_2(z,z)) \ \longrightarrow\ \{5/z\}(\pi_1(\pi_2(z,z))) \equiv \pi_1(\pi_2((5,4),(5,4)))$$

Substitution has removed the relation between the value $(5,4)$ and the variable z; in the resulting term we no longer know where $(5,4)$ originated. Immediately we are presented with a glaring incompatibility between the usual reduction strategy for CBV and dynamic update. In the situation posed it is hopeless to try to give meaning to a notion of upgrading z, as z no longer 'exists'. If we are to make sense of dynamic rebinding, as we must if we are to make progress in understanding dynamic updating, then we need a reduction strategy that preserves the structure of the computation, allowing us to refer back to the results of previous computations and delaying the collapse of the indirection from variable names to their bound values.

One way of presenting such a system without introducing any extra syntactic constructors is to do away with substitution and allow reduction under lets that bind values, which we call *value-lets*. For example, instead of

$$\textbf{let}\ x = 1\ \textbf{in}\ \textbf{let}\ y = 2\ \textbf{in}\ \pi_1(3,4) \ \longrightarrow\ \textbf{let}\ y = 2\ \textbf{in}\ \pi_1(3,4)$$

we could allow

$$\textbf{let}\ x = 1\ \textbf{in}\ \textbf{let}\ y = 2\ \textbf{in}\ \pi_1(3,4) \ \longrightarrow\ \textbf{let}\ x = 1\ \textbf{in}\ \textbf{let}\ y = 2\ \textbf{in}\ 3$$

as the lets bind values. A consequence of such a theory is that we eventually reach a variable during reduction, for example:

$$\textbf{let}\ z = (5,4)\ \textbf{in}\ \pi_1(\pi_2(z,z))$$

where the leftmost z must be resolved as it is in *redex* position. Of course, we are not stuck, because the answer is contained in the surrounding context which contains a let-binding for z. All we have to do is look it up:

$$\mathbf{let}\ z = (5,4)\ \mathbf{in}\ \pi_1(\pi_2(z,z)) \ \rightarrow\ \mathbf{let}\ z = (5,4)\ \mathbf{in}\ \pi_1(\pi_2((5,4),z))$$

Or do we? Well, it is certainly valid to instantiate the variable z at this point, but it is not *necessary*. The value of z is not actually needed, because the inner π_2 projection does not inspect the structure of z. Performing the inner projection leads to

$$\mathbf{let}\ z = (5,4)\ \mathbf{in}\ \pi_1 z$$

The ensuing projection presents a problem as it does rely on the structure of z – we cannot project from a variable, only a tuple. We are forced to instantiate the z and say that z is in *destruct position* as the value z binds is about the destructed (decomposed):

$$\mathbf{let}\ z = (5,4)\ \mathbf{in}\ \pi_1 z \ \rightarrow\ \mathbf{let}\ z = (5,4)\ \mathbf{in}\ \pi_1(5,4) \ \rightarrow\ \mathbf{let}\ z = (5,4)\ \mathbf{in}\ 5$$

In the discussion above we proposed allowing execution to continue under value-binding lets and identified two ways to delay the instantiation of variables. The first *delayed instantiation* strategy, which we call *redex time*, instantiates (looks up) variables when they enter redex position in a usual left to right CBV evaluation order. The second strategy, which we call *destruct time*, instantiates variables later when a destructive operation is directly applied. A destructive operation is one that needs to look at the value bound to a variable such as project or function application.

Either of these two instantiation strategies are suitable for use in dynamic environments. Both delay instantiation of variables, preserving the link between binder and bindee and thus, in the presence of updates, allow programs to use the most recent version of a bound variable.

Destruct-time instantiation preserves the binder-bindee link longer, making the program aware of more updates, while redex-time instantiation is a more familiar model (and may carry more intuition). Chapter 2 formalises these reduction strategies and defines a notion of update within the calculus.

1.2.1 CBV Equivalence

We would like to continue using our usual CBV reasoning and intuition, therefore we hope that the calculi we describe are, in the absence of rebinding or update, equivalent to the usual CBV semantics. The intuitive reason for believing that they still operate 'by value' (and thus CBV reasoning still applies) is that instantiation only occurs for value bindings, and evaluation only continues under a let once that let binds a value. Slightly more formally, observe that substituting away the lets from the delayed term results in a term that matches substitution-based reduction:

$$\begin{tikzcd}
\begin{array}{l}\mathbf{let}\ f = \lambda x.e\ \mathbf{in} \\ \quad f\ 3\end{array} \arrow[r, "\text{delayed inst.}"] \arrow[d, "\text{Sub. out lets}"] & \begin{array}{l}\mathbf{let}\ f = \lambda x.e\ \mathbf{in} \\ \quad (\lambda x.e)\ 3\end{array} \arrow[r, "\text{app}"] \arrow[d, "\text{Sub. out lets}"] & \begin{array}{l}\mathbf{let}\ f = \lambda x.e\ \mathbf{in} \\ \mathbf{let}\ x = 3\ \mathbf{in} \\ \quad e\end{array} \arrow[d, "\text{Sub. out lets}"] \\
(\lambda x.e)\ 3 \arrow[r, equal] & (\lambda x.e)\ 3 \arrow[r, "\text{app}"] & \{3/x\}e
\end{tikzcd}$$

However, the situation is more subtle in the presence of a recursive definitions (letrec), as the substitution may unfold the recursion. For example, the reason that the central substitution of lets results in $(\lambda x.e)\ 3$ and not $(\lambda x.\{\lambda x.e/f\}e)\ 3$ is that f does not appear free in e.

Although a correspondence as described above shows that reduction sequences are related, it says nothing about termination: there may be programs that terminate under the usual CBV reduction but diverge under our delayed instantiation approach, or vice versa. Pleasingly, this turns out not to be the case. In fact, the contextual equivalence relations of delayed instantiation calculi coincide with that of the usual CBV, in a sense to be made clear in Chapter 3.

1.2.2 Safe Dynamic Relinking

As discussed, dynamic relinking involves changing the value associated with some binding available to the program. In its simplest form this could be a non-deterministic change to a variable in the program, in which case our only job would be to place a restriction on when such changes could occur, as without such restrictions it would be hard for a programmer to reason about evaluation. However, here we have made the tacit assumption that the variable retains its type, forcing the relinked value to be of the same type as (or possibly a subtype of) the old one — a far too restrictive constraint if we want to use the relinking mechanism to perform dynamic updates. Of course, if we permit types to be changed then we must transform any values associated with that type accordingly, otherwise we may end up with a type-inconsistent program.

One way to enforce type consistency is to require that the newly-relinked program type check in its entirety (e.g. as in Hicks' thesis [Hic01]). However, with this approach the program requires a runtime representation that is checkable, such as typed assembly language [MCG+99] combined with disassembly, proof-carrying code [Nec97] used in the same way, or some higher-level typed representation such as Java bytecode, which is not appropriate for all applications. Worse though, is that the programmer is left to guess how to produce a patch that when applied to the program will produce a typeable program, and moreover, whether or not the program is typeable depends on when the patch is applied. Thus, even when we have a way to talk about relinking in our language, we are still left wondering about the inter-related problems of how to change the type of bindings and how to reliably modify the state to be consistent with this in a way that is both natural and intuitive to the programmer.

1.3 Dynamic Update

Now that we have some understanding of delayed instantiation and dynamic relinking, the core challenge of changing the representation of data can be addressed. For our purposes the representation of data can be seen as its type; all items of a given type having the same representation and each type's representation being unique.

Suppose, for a moment, that all data is allowed to change type. Further assume that we identify the type of data by tagging it with its type T, written $[-]_T$. Consider $[1]_{\mathsf{int}} + [2]_{\mathsf{int}}$. If we change the representation of $[1]_{\mathsf{int}}$ to $[\text{“1”}]_{\mathsf{string}}$ then this will only make sense if we change the definition of $+$ and possibly the representation of $[2]_{\mathsf{int}}$ to match. One possibility is to change $[2]_{\mathsf{int}}$ to $[\text{“2”}]_{\mathsf{string}}$ and the definition of $+$ to a function that acts on strings, for example, concatenation. This approach is not very practical for two reasons. First, there is no way to identify the data item to change (we have no name for $[1]_{\mathsf{int}}$) and second, each data item in a program needs changing on an individual basis – the update mechanism lacks structure. The first of these problems has an obvious solution, which is to allow only data bound by a let to be changed (although this is not without its problems due to alpha conversion). This still leaves the programmer to search out all variables that should be changed, suggesting that we need a higher-level notion of change.

When programming in a typed language, groups of variables that have similar meaning to the programmer, such as floating point numbers that denote money or a record structure that denotes an address, are usually grouped together using a *named type*, for example, $\mathsf{money} = \mathsf{float}$ and $\mathsf{address} = \{house\_no : \mathsf{int}, postcode : \mathsf{string}\}$. These named types, being assigned by the programmer, create logical divisions of the data where all members of a particular division (named type) are likely to change representation together; it is preferable to have all monetary values as floats or strings, but not a mix of representations. Prompted by this observation we do not concentrate on changing the type of individual data items, but on changing the definition of named types. A safe update will ensure that such a change does not invalidate the type structure of the program.

Consider the following simple program

$$\begin{aligned}&\mathbf{type}\ \mathsf{t} = \mathsf{int}\\&\mathbf{let}\ x : \mathsf{t} = 5\ \mathbf{in}\\&\quad x \div x\end{aligned}$$

If we try to update t to string, or some other exotic type, then the data bound to x must be transformed accordingly. However, such a transformation will invalidate the subsequent division operation — especially as it is integer division. We clearly need some control over *what* can be updated, and because the safety of an update is flow-dependent (e.g. after the division is performed there is no constraint on the updatability of x), we also need to consider *when* updates are valid.

In the last example, the unsoundness introduced by updating type t was due to the fact that x, of type t, was subsequently used at t's originally defined type. This is unsound because the type of a variable represents an assumption about properties of the data associated with that variable. If we change the type then we change the assumptions, and continuing to use old, now invalid, assumptions will lead to error. If the type of a variable is to be safely updated it seems necessary to require that the variable never be used at the old type following the update. With no further refinement this condition would mean the variable could never be used concretely in the execution of the rest of the program, a limiting restriction indeed. This motivates a second natural requirement, that every function that uses values of type t concretely be replaced with a function that acts on those values at the new type, i.e. is consistent with the new definition of t. With this in place, it is only the code on the call stack that must not use the variable, for this usage will be at the old type. It is perfectly safe to pass values of type t as parameters in function calls, as the function is guaranteed to have been replaced and so consistent with the newly updated type.

There are many outstanding questions. How do we know if a given update is safe? How do we find and convert data of type t at runtime? How do we know when data of named type is used concretely? At what point do we know if a type is updatable – runtime or compile time? We outline our solutions in subsequent sections.

1.3.1 After the Event Abstraction

Having decided to focus on updating the definition of named types, we face a problem: if we have the type definition $\mathsf{t} = \mathsf{int}$ how can we distinguish between data of type t and that of type int? This is important as when updating t we want to change all data of type t, which is not the same as every integer. One way, and the one we shall investigate, is to reflect the difference in the term structure. We let $\mathbf{abs}_{\mathsf{t}}\ 3$ be of type t with the interpretation that 3 is *abstracted* to type t, while 3 is of type int. Of course, we need a way to *concretize* a value of named type back to its underlying type, which we write $\mathbf{con}_{\mathsf{t}}\ e$ with the semantics that $\mathbf{con}_{\mathsf{t}}\ (\mathbf{abs}_{\mathsf{t}}\ 3) \rightarrow 3$.

There are two benefits to this explicit representation. First, it is immediate exactly which functions use data of a named type concretely, as those contain a subterm of the form $\mathbf{con}_{\mathsf{t}}\ e$. Second, when we change the definition of t we can easily locate data of type t, as they are subterms of the form $\mathbf{abs}_{\mathsf{t}}\ e$. The downside is that the programmer must insert these explicit coercions into and out of the named type as t and int are distinct, whereas the usual semantics for named type definitions implies that t and int can be used interchangeably whenever the declaration $\mathbf{type}\ \mathsf{t} = \mathsf{int}$ holds. Luckily, we can (almost[2]) regain the usual semantics for the programmer by inserting these coercions automatically, as we show in Section 5.2. For now we will consider programs with explicit coercions.

To see how the explicit coercions work in practice suppose we have two named types defined, $\mathsf{t} = \mathsf{int}$ and $\mathsf{s} = \mathsf{int}$, and consider the following program

$$\begin{array}{l}
\mathbf{fun}\ \mathrm{double}(x : \mathsf{t}) : \mathsf{int} = 2 * (\mathbf{con}_{\mathsf{t}}\ x)\ \mathbf{in} \\
\mathbf{let}\ x : \mathsf{t} = \mathbf{abs}_{\mathsf{t}}\ 2\ \mathbf{in} \\
\mathbf{let}\ y : \mathsf{s} = \mathbf{abs}_{\mathsf{s}}\ 3\ \mathbf{in} \\
\mathbf{let}\ z : \mathsf{int} = (\mathbf{con}_{\mathsf{t}}\ x) + (\mathbf{con}_{\mathsf{s}}\ y)\ \mathbf{in} \\
\quad z + \mathrm{double}(x) + (\mathbf{con}_{\mathsf{s}}\ y)
\end{array}$$

At the start of execution neither t nor s are updatable because the continuation of the process contains concretions for both types. Let us use substitution for $\mathbf{let}$s and (redex-

[2]The type definition $\mathbf{type}\ \mathsf{t} = \tau$ will be transparent everywhere except at reference types

time) instantiation for functions, so that substituting away both **let**s gives

$$\begin{array}{l}\mathbf{fun}\ \mathrm{double}(x:\mathsf{t}):\mathsf{int}=2*(\mathbf{con}_{\mathsf{t}}\ x)\ \mathbf{in}\\ \mathbf{let}\ z:\mathsf{int}=(\mathbf{con}_{\mathsf{t}}\ (\mathbf{abs}_{\mathsf{t}}\ 2))+(\mathbf{con}_{\mathsf{s}}\ (\mathbf{abs}_{\mathsf{s}}\ 3))\ \mathbf{in}\\ \quad z+\mathrm{double}(\mathbf{abs}_{\mathsf{t}}\ 2)+(\mathbf{con}_{\mathsf{s}}\ (\mathbf{abs}_{\mathsf{s}}\ 3))\end{array}$$

whereupon the **abs** and **con** annihilate each other, the program reducing to

$$\begin{array}{l}\mathbf{fun}\ \mathrm{double}(x:\mathsf{t}):\mathsf{int}=2*(\mathbf{con}_{\mathsf{t}}\ x)\ \mathbf{in}\\ \mathbf{let}\ z:\mathsf{int}=2+3\ \mathbf{in}\\ \quad z+\mathrm{double}(\mathbf{abs}_{\mathsf{t}}\ 2)+(\mathbf{con}_{\mathsf{s}}\ (\mathbf{abs}_{\mathsf{s}}\ 3))\end{array}$$

The resulting program is updatable for t, but not for s. This is because the continuation of the program will directly use s concretely, but t will only be used concretely indirectly through the function double, which will be replaced on any update to t to be consistent with the new definition. To update t to, say, the record $\{a : \mathsf{int}, b : \mathsf{int}\}$ we provide a coercion function $c : \mathsf{int} \to \{a : \mathsf{int}, b : \mathsf{int}\}$, which the runtime system can easily insert at every abstraction. Consistent with the requirements discussed earlier we also replace the function double to act consistently with the new definition of t. The resulting program is

$$\begin{array}{l}\mathbf{fun}\ \mathrm{double}(x:\mathsf{t}):\mathsf{int}=2*(\mathbf{con}_{\mathsf{t}}\ x).a\ \mathbf{in}\\ \mathbf{let}\ z:\mathsf{int}=2+3\ \mathbf{in}\\ \quad z+\mathrm{double}(\mathbf{abs}_{\mathsf{t}}\ (c\ 2))+(\mathbf{con}_{\mathsf{s}}\ (\mathbf{abs}_{\mathsf{s}}\ 3))\end{array}$$

In this example we have assumed updates to be asynchronous, meaning that we can attempt to apply them without regard to the program's state. The formalism described in this thesis uses synchronous updates, their possibility being explicitly marked with the keyword **update**. Asynchronous updates are hard to reason about because they can happen in any program state, but in contrast synchronous updates allow one to know some information about the state of the program at the time the update is applied.

1.3.2 A Capability Approach

The machinery discussed thus far allows for program patches that can change the definition of named types in a way that is both safe and whose application is automatic. While this is an advance over previous systems, it still has one flaw: when the initial program is run we have no guarantee as to the updatability of any type. We only discover if an update is possible when we attempt to apply it. We would prefer to know whether each type is updatable in advance of execution; if our program is not available for update then we have precluded the very thing we set out to achieve. Our approach to this problem will be to use the intuitive dynamic notion of capability to develop a capability-based type system that predicts which types are modifiable at each update point statically. This static system is developed in Chapter 6 and here we restrict our discussion to explaining the intuitive notion of capability that lies behind it.

Updates can be made safe through a check at update time, as proposed in the previous section, but suppose for a moment that no such check was in-place. How could we detect that an invalid update had lead to an error dynamically? An approach we shall describe is based on the notion of *capability*.

Suppose we augment program configurations with a capability denoted Δ to have configurations $(\Delta; P)$, where Δ is a set of named types and P is the program. The notion of capability we shall use is that whenever $\mathrm{t} \in \Delta$ the program P has the capability to concretize data of type t.

Assume that initial configurations have the ability to concretize all program types. We must say how execution maintains the capability set so as the set always contains exactly the types the program can concretize. We now look at how this capability set is used and maintained.

Concretization This is where errors are detected. A fundamental constraint is that a program only concretes at a type t if it has the capability to do so. By this we mean that the reduction $\Delta; \mathbf{con}_{\mathrm{t}}\ (\mathbf{abs}_{\mathrm{t}}\ v) \rightarrow \Delta; v$ is only possible whenever $\mathrm{t} \in \Delta$. Whenever we reach a concretion and $\mathrm{t} \notin \Delta$ then an error occurs as we do not have the capability to proceed.

Updates Whenever an update occurs to a named type t the capability to concretize at that type needs to be revoked:

$$\Delta; P \xrightarrow{\text{update to t}} \Delta \backslash \{\mathrm{t}\}; P'$$

where P' is the updated program with the data of type t transformed to be consistent with the new definition.

Function application Updates reduce our capability and concretions can lead to errors if our capability is not big enough. If these were the only rules our program would become less and less updatable as time progressed. Fortunately they are not. It is our requirement to replace functions that use a type begin updated concretely that plays a crucial rôle here. Because of this requirement, whenever a function is called it is guaranteed to use the new definition of t. Thus even if the calling context does not have the capability to concretize at t, this does not preclude the called function from concretising at t. Thus, we are free to expand the capability when calling a function. We must remember through, to reduce the capability upon returning from the function, as then we will be back in old code that assumes the old definition of t.

1.4 Pragmatic Update

A reasonable question to ask is whether the theory we develop has any practical value. This can only be answered through the experience of using a real system, for which a significant start has been made. The theory presented in this thesis has been implemented for the C programming language in the form of a prototype source-to-source compiler for turning regular C programs into updatable ones and for compiling dynamic patches that can be applied by a simple run-time system. We accept all C programs, marking types non-updatable where we must be conservative due to C's unsafe features.

Chapter 7 describes a prototype implementation of the theory as a sequence of source-to-source transforms on C programs together with constraint solving to determine a program's updatability. Additionally, it extends the theory to cope with specific C language features and gives some preliminary performance results.

1.5 Update Systems Introduced in This Thesis

This thesis introduces the four update systems, Lambda-update, The Update Calculus, Proteus and $\mathrm{Proteus}^{\Delta}$. They increase in sophistication, following the development outlined in this introduction.

1.5.1 Lambda-Update

This is a simple extension of the delayed instantiation calculus λ_d. It provides a straightforward mechanism for update that allows one to change any binding in scope at the reduction of the update. It is typesafe and flexible, but achieves this via typechecking at runtime.

1.5.2 Update Calculus

This applies the ideas of λ_r in a simple setting, giving semantics to a statically typed Erlang-like language. This language requires just as many dynamic checks as Lambda-Update, but is more practical, showing how the ideas can apply at the module level.

1.5.3 Proteus

Proteus starts to remove the dynamic checks required by the previous calculi. Proteus replaces a runtime type check with a simple syntactic check, which is both simple to verify and obviates the need for runtime type information. It still allows types to change, but places a restriction that only named types can change. In short, the relevant part of the type system is reified in the syntax and only a check for the presence of certain syntactic forms in the continuation is required at update time.

1.5.4 Proteus$^{\Delta}$

$\mathrm{Proteus}^{\Delta}$ is an extension to Proteus that completely removes the need to examine code at runtime. Through a static analysis, each update point is (conservatively) annotated with the types that may not be changed by updates applied at that update point. The only runtime check is to compare the types changed by the update to those prohibited from change at the active update point. Any other checks (such as type checking new code) can be done offline. Because the analysis is conservative, $\mathrm{Proteus}^{\Delta}$ accepts a potentially smaller class of updates than Proteus. However, the analysis can be used promiscuously to discover program points at which more types can be updated than any other. Ideally we will find a universal point that admits updates to all types, or a set of update points that collectively allow all types to be updated.

1.6 Related Work

This section relates our work to the existing literature. Section 1.6.1 reviews work related to our development of a delayed instantiation semantics for the λ-calculus, Section 1.6.2 details the prior work on dynamic update and Section 1.6.3 reviews previous work on capability type systems.

1.6.1 λ and Delayed Instantiation

Our approach to the semantics of the CBV λ-calculus retains bindings for later instantiation and in this respect has some similarities with prior work on explicit substitutions [ACCL90], sharing in call-by-need languages [AFM$^+$95], and work on the compilation of extended recursion [HLW03, Hir03].

The work on explicit substitutions presents the $\lambda\sigma$-calculus, a call-by-name (CBN) λ-calculus that moves the usual notion of substitution from the meta-level into the term structure. Although their evaluation strategies are CBN, the obvious presentation of a CBV strategy would lead to λ_r (redex-time) instantiation of variables.

The Call-by-Need Lambda Calculus [AFM$^+$95] of Maraist, Odersky and Wadler uses lets to share the results of computations. Instantiation from surrounding let bindings is performed, although they are lazy and so the timing is not specified. Their computations result in "answers" that are similar to our value forms. They also show a correspondence to the call-by-name lambda calculus using operational reasoning.

The idea of incorporating a destruct-time instantiation strategy in a CBV λ-calculus, as we use in λ_d, was simultaneously discovered by Hirschowitz et al. [HLW03] in the setting of extending mutual recursion beyond data of function type in CBV languages. They prove a correspondence with an allocation-style semantics, while we prove a correspondence with a substitution semantics.

Although the instantiation strategies of our work and those mentioned above are similar, differences occur in how the environment is represented. In particular, we preserved the structure of the environment whereas the others do not. The $\lambda\sigma$-calculus percolates substitutions through the term structure to the variables, whereas the the calculus presented by Hirschowitz does the opposite, floating value-binding lets up to the top-level, after which instantiation is possible. The call-by-need lambda-calculus allows its value binding lets to be flattened and garbage collected.

1.6.2 Dynamic Update

DSU has been studied in both industry and academia for many years resulting in a number of implementations. Surprisingly, perhaps, there has been little formal study and many of the implementations offer little in the way of safety guarantees. The implementations that do give some safety guarantees do so with severe restrictions on what can be updated. Previous work can be categorised into three approaches: those that encode DSU into an existing language via the use of design patterns; those that take a systems approach; and those that incorporate it as a language feature. Our approach is the latter.

Coding up DSU

We are aware of two approaches[HG98, SC05] to encode DSU in an existing (unmodified) language. The first gives a way to change the implementation of classes in C++ and the second describes a way to build dynamic applications in Haskell [SC05].

Work on dynamic C++ classes [HG98] proposes the use of proxy classes to provide a level of indirection between the current implementation for an object and its interface. This means the interface used by the rest of the program can remain fixed while the underlying implementation changes. Of course, this means that extra fields and methods cannot be added. After a new implementation of a class is provided, all subsequently created objects are instances of this new class. However, no attempt is made to upgrade existing objects that use the old implementation. The approach does complicate the use of inheritance (due to the separation of interface and implementation) and dynamic objects have to be created through a factory pattern.

Stewart and Chakravarty [SC05] describe a method for writing applications in Haskell so that code can be *hot swapped*. They advocate building applications around a small static core that interacts with both the dynamic linker, to reload components, and with the dynamic part of the program. The entry point of the program's dynamic part is parametrised by the state that needs to be preserved through an upgrade. Little support is offered for dealing with changes to the representation of data and their solution for changing the state type is to serialise the data, convert it and reinject. This happens outside the type system and is thus unchecked.

MagicBeans [CEM04] is an software engineering solution to dynamic update that employes a plugin architecture. It uses the reflection, custom loading and dynamic linking capabilities of Java to implement DSU as a collection of design patterns. This allows one to write applications that can be extended at runtime, but the extent and direction of this expansion must have been anticipated at design time. Nonetheless, it does provide a limited form of DSU.

Barr and Eisenbach [BE03] develop a distributed framework for managing the upgrade of application components. This works in the existing Java framework and so has the limitation that updated components must be binary compatible [DWE98, DEW99] with those they replace. Their system checks components to ensure that this requirement is met.

Systems approach

There are three approaches that we group together as systems approaches: cluster rolling upgrade, process migration, and OS controlled.

Cluster Rolling Upgrade The cluster rolling upgrade approach is simple and works well, but only for a limited class of client-server applications where request can be fed to multiple instances of the server by a proxy. As long as the process of upgrading keeps one server alive, the clients will not suffer any break in service. While simple, this process requires redundant hardware (if the OS is to be upgraded) which may not be available (e.g. when upgrading a personal operating system or an embedded device) and requires the program to have a strict client-server architecture that is stateless so that different requests from a given session can be distributed among multiple servers.

Process Migration An issue that DSU could address, and a particular issue the systems community has focussed on, is that of updating an operating system while its processes remain running. The common approach taken by the systems community is process migration [Smi88]. Whenever an OS needs to be upgraded its processes can be checkpointed [Pla97, BMP$^+$04], a process whereby the memory footprint is serialised to storage. This image can then be transmitted to another machine where it can be restarted. Meanwhile the original machine's OS can undergo an upgrade and be rebooted. There are issues here with what happens to resources acquired before checkpointing and whether they will be available on the target machine. Also, this approach does not deal with the problem of state transfer between the upgraded process and the migrated one. It is worth pointing out that this can be done without redundant hardware using virtualisation [LSS04].

OS Controlled Most modern operating systems support some form of dynamic code loading at both the user and kernel level. For example, Linux supports `dlopen(3)` for user-level dynamic linking and kernel modules for extending the operating system at runtime. These approaches only allow for dynamic linking, not the relinking and state transfer needed for DSU. The only operating system we are aware of to support DSU is the K42 research operating system [BAS$^+$05, SAH$^+$03] developed at IBM Research. In K42 components must be quiesced before they are updated, which is achieved by blocking new calls while waiting for existing ones to complete. A few protocols of state transfer are supported, although the actual process is left to the programmer. In contrast, we allow the update of active code and provide a general method for state transfer. K42 is a multi-threaded OS and thus supports DSU in this context, however, threads have to be blocked for update.

DSU as a Language Feature

Research into DSU has been mainly focused at the language level, where research efforts may usefully be assessed in three directions. First, how available the system is for update, by which we mean does it restrict update to certain parts of the program or only allow it at certain points during execution. Second, how they deal with the state transfer problem and the degree to which conversion of data at runtime is automated, as this is the most difficult and error prone part of this approach to DSU. Third, whether any guarantees of safety are made.

Much of the early research allows code to be replaced, but is not type safe [FS91, HG98, Gup94, BH00] while Erlang [AVWW96, AV91], which has been successfully used in industry, can generate dynamic errors. The closest systems to ours are those by Hicks [Hic01], Gilmore et al. [GKW97], Duggan [Dug01], and work on adding DSU to Java [Dmi01b, ORH02, MPG$^+$00, BLS$^+$03].

Hicks' system adds a synchronous update feature to a type-safe C-like language with garbage collection. Safety is provided by building the system on top of TAL, a typed assembly language. Updates in this system are patches consisting of a state transformer, stub functions and replacement functions. The state transformer function is a user supplied procedure that runs directly after new code is installed and is intended to convert the old state to be consistent with the new code. Stub functions allow the type of functions to change by proxying calls from old code, while new code calls the new functions directly. Hicks' system allows user-defined types to change, but relies on the user to manually locate and transfer data between them, whereas we support named type changes with automatic conversion. Hicks' system is also available for update only when the call stack is empty, whereas we support update with a non-empty call stack.

We use a similar type-based approach to Duggan although our presentation is more elementary. Duggan allows multiple versions of a type to coexist simultaneously with conversions between them being applied automatically. In contrast, our system maintains a single definition for each named type. We believe that our *representation consistent* approach makes it simpler for a programmer to reason about their program because they only need consider the latest definition of a named type. Our systems also differ in how data conversion is implemented. While Duggan's semantics check a runtime type tag, we avoid this complication by expressing an update as a rewrite to the program.

Gilmore's Dynamic ML system allows dynamic update of ML modules. The types exported by updatable modules are restricted to abstract types, the replacing module must be a sub-signature of the existing module, and replacement can only occur when the module is quiescent, that is, none of its functions are on the call stack. It allows the abstract types of a module to be changed, applying a user-supplied conversion function automatically. This conversion process is specified as part of the garbage collector which is convenient for implementation but results in the semantics of update being given in a somewhat indirect way. While it is not hard to believe this approach to be sound (although this is not proved), it imposes a serious restriction on expressiveness of the updates and on coding style.

A few researchers have proposed to incorporate DSU into Java [GJSB00]. Dimitriev [Dmi01b, Dmi01a] studied the state transformation problem from the point of view of persistent Java programs, where databases of persistent objects need to evolve with the program. This is a simpler problem than state transfer in DSU as the program is not running when an update is applied to the database. Dimitriev also modifies the Java Virtual Machine to support the replacement of class implementations with binary compatible ones (see [DWE98] for a discussion of binary compatibility).

Orso et al. [ORH02] propose a technique similar to Dynamic C++ for use with Java. The use of templates in Dynamic C++ is replaced with a tool that rewrites Java code to explicitly support DSU. As with Dynamic C++, the interface to classes must remain fixed and in addition, reflection and native methods are not compatible with the technique.

Dynamic Java Classes [MPG^{+}00] extend the Java class loader to support (nearly) arbitrary changes to classes. To maintain type safety, whenever a class is updated all other classes are examined to see if they depend on the class being replaced. If they do then the system requires them to be replaced as well. All existing objects of an updated class are converted at update time during garbage collection along similar lines to Dynamic ML. In contrast, our system spreads out the cost of instance conversion rather than introduce a possibly lengthy delay at update time. We also give a formal semantics along with proofs of correctness.

Drossopoulou and Eisenbach [DE03] formalise a system of dynamic relinking, intended as an extension to the dynamic linking already supported in languages such as Java and C#. While dynamic relinking does not constitute DSU on its own, it is an essential ingredient. The approach is quite flexible, differentiating between the loading of the class and class members. This enables method signatures and method bodies to be loaded separately, so that choices about the exact implementation are delayed as long as possible. In return for this flexibility extensive type checking is required at runtime.

Boyapati et al. [BLS+03] develop a system for online upgrades to code and object representation in a persistent object store that supports local reasoning about objects. They, like us, transform objects lazily as they are accessed to minimise interruption, and allow other upgrades to begin before previous ones have completed. Upgrades are performed by conversion functions which must run in a specified order to maintain type safety (earlier updates must be applied before later ones). They ensure the ordering by imposing the encapsulation constraint that every object a given object depends on must be an encapsulated subobject. When this constraint fails the decision is passed to the programmer, although some support is given to automate the common cases.

None of the systems mentioned above give guarantees of updatability as we provide in Chapter 6.

1.6.3 Capability Type Systems

The notion of capability has a long history in computer science (for example [WN79]). More recently the notion has been used in the static analysis of programs, first introduced by Walker, Crary and Morrisett [WCM00] and used extensively in [DF01, FD02] for statically checking resource management and in [Wal00] for enforcing security policies. We are the first to apply capability type systems to solve the DSU problem.

1.7 Collaboration

Collaboration is an important part of research and this thesis is no exception. Chapters 2 and 3 are the result of a joint project previously published in [BHS+03a] with Peter Sewell, Gavin Bierman, Mike Hicks and Keith Wansbrough, although Chapter 3 is entirely my own work. Chapter 4 contains my contribution to work previously published in [BHSS03] with Peter Sewell, Gavin Bierman and Michael Hicks. Chapters 5 and 6 have previously been published in [SHB+05] with Michael Hicks, Gavin Bierman, Peter Sewell and Iulian Nemtiu. I took the lead rôle in this research, but Michael was the first to sketch the ideas of the capability type system. Finally, the implementation reported in Chapter 7 was joint work with Iulian Nemtiu, Mike Hicks and Manuel Oriol, although I developed the abstraction violating alias analysis, was a major contributor to the implementation and the presentation I give is, of course, my own.

Part I

The λ-calculus and updatability

2

Delayed instantiation

Most programming languages choose static binding, where free variables are bound by the enclosing binders in their source-code context. This is for good reason, as it allows static type systems to be employed so that errors may be caught early and identifiers can be resolved at compile time, which is good for efficiency. On the other hand, dynamic binding is obligatory for systems that support dynamic update: new code must be loaded into the system which will want to make use of existing bindings in the program. Dynamic binding is also required to support many other language features, such as marshalling. For example, if a computation is to be sent across a network then resources bound at one site may need to be rebound at another.

The standard Call By Value (CBV) semantics for the lambda calculus provides an inadequate model in which to consider dynamic binding and consequently is hard to extend with dynamic update. This chapter exposes its deficiencies and forms a theoretical foundation for dynamic update by re-examining the standard operational semantics for the CBV lambda calculus. We change the reduction strategy, but remain faithful to the traditional CBV semantics (in a sense to be made clear in the next chapter). At the end of the chapter we show how our semantics can be easily extended to give a calculus that supports dynamic update.

Construct-time λ_c

Values	v	$::=$	$n \mid () \mid (v, v') \mid \lambda z{:}\tau.e$
Atomic evaluation contexts	A	$::=$	$(\_, e) \mid (v, \_) \mid \pi_r\ \_ \mid \_\ e \mid v\ \_ \mid$ **let** $z = \_$ **in** e
Evaluation contexts	E	$::=$	$\_ \mid E.A$

Reduction Rules

(proj)	$\pi_r(v_1, v_2) \longrightarrow v_r$
(app)	$(\lambda z{:}\tau.e)v \longrightarrow \{v/z\}e$
(let)	**let** $z = v$ **in** $e \longrightarrow \{v/z\}e$
(letrec)	**letrec** $z = \lambda x{:}\tau.e$ **in** $e' \longrightarrow \{\lambda x{:}\tau.$**letrec** $z = \lambda x{:}\tau.e$ **in** $e/z\}e'$ if $z \neq x$
(cong)	$\dfrac{e \longrightarrow e'}{E.e \longrightarrow E.e'}$

Error Rules

(proj-err)	$E.(\pi_r\ v)\mathsf{err}$	if not exists v_1, v_2 such that $v = (v_1, v_2)$
(app-err)	$E.(v'\ v)\mathsf{err}$	if not exists $(\lambda z{:}\tau.e)$ such that $v' = \lambda z{:}\tau.e$

Figure 2.1: Standard Call-by-Value Lambda Calculus

2.1 The failure of CBV semantics

Consider the CBV λ-calculus, and in particular the way in which identifiers are instantiated. The usual operational semantics, recalled in Figure 2.1, substitutes out binders – the standard *construct-time* (app) and (let) rules

(app)	$(\lambda z{:}\tau.e)v$	$\longrightarrow$	$\{v/z\}e$
(let)	**let** $z{:}\tau = v$ **in** e	$\longrightarrow$	$\{v/z\}e$

instantiate all instances of z as soon as the value v it has been bound to has been constructed. This semantics is not compatible with dynamic update, as it loses too much information. To see this, suppose that the reduction of e in **let** $z = v$ **in** e is allowed to accept dynamic updates to the program. More often than not the update will want access to the surrounding state, either to use it or to modify it. With the (let) rule this would be futile, as the z is substituted away before the execution of e even begins, let alone an update point is reached.

We therefore need a more refined semantics that preserves information about the binding structure of terms, allowing us to delay 'looking up' the value associated with an identifier as long as possible so as to obtain the most recent version of its definition.

This should maintain the essentially call-by-value nature of the calculus (we elaborate below on exactly what this means).

We present two reduction strategies with delayed instantiation in §2.2. The *redex-time* (λ_r) semantics resolves identifiers when in redex position. While this is clean and simple, it is still unnecessarily eager, and so we formulate the *destruct-time* (λ_d) semantics to delay resolving identifiers until their values must be destructed. After establishing these basic calculi, we extend the latter one to support dynamic update. As an example, consider the expression on the left below:

$$\begin{array}{lcl} \textbf{let}\ x = 5\ \ \textbf{in} & \xrightarrow{\{y \Leftarrow (x,6)\}} & \textbf{let}\ x = 5\ \ \textbf{in} \\ \textbf{let}\ y = (4,6)\ \textbf{in} & & \textbf{let}\ y = (x,6)\ \textbf{in} \\ \textbf{let}\ z = \textbf{update}\ \ \textbf{in} & & \textbf{let}\ z = ()\ \textbf{in} \\ \pi_1\ y & & \pi_1\ y \end{array}$$

Let the **update** expression indicate that an update is possible at the point during evaluation when **update** appears in redex position. At that run-time point the user can supply an update of the form $\{w \Leftarrow e\}$, indicating that w should be rebound to expression e. In the example this update is $\{y \Leftarrow (x,6)\}$; the let-binder for y_1 is modified accordingly yielding the expression on the right above, and thence a final result of 5. For a reduction like this to even be possible, it requires that let binders are not substituted away, but that execution continues under them.

2.2 CBV λ-calculus revisited

This section defines the late instantiation calculi λ_r and λ_d. We take a standard syntax:

$$\begin{array}{llcl} \text{Identifiers} & x, y, z & & \\ \text{Integers} & n & & \\ \text{Types} & \tau & ::= & \mathsf{int} \mid \mathsf{unit} \mid \tau * \tau' \mid \tau \rightarrow \tau' \\ \text{Expressions} & e & ::= & z \mid n \mid () \mid (e, e') \mid \pi_r\ e \\ & & \mid & \lambda z{:}\tau.e \mid e\ e' \mid \textbf{let}\ z = e\ \ \textbf{in}\ \ e' \\ & & \mid & \textbf{letrec}\ z = \lambda x{:}\tau.e\ \ \textbf{in}\ \ e' \end{array}$$

where r ranges over $\{1, 2\}$. Expressions are taken up to alpha equivalence (though contexts are not). It is simply-typed, with a standard typing judgement $\Gamma \vdash e{:}\tau$ defined in Figure 2.2, where Γ ranges over sequences of $z{:}\tau$ pairs containing at most one such for any z.

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

$$\frac{}{\Gamma, z{:}\tau, \Gamma' \vdash z{:}\tau}$$

$$\frac{}{\begin{array}{c}\Gamma \vdash n{:}\mathsf{int}\\ \Gamma \vdash (){:}\mathsf{unit}\end{array}} \qquad \frac{\begin{array}{c}\Gamma \vdash e{:}\tau\\ \Gamma \vdash e'{:}\tau'\end{array}}{\Gamma \vdash (e, e'){:}\tau * \tau'} \qquad \frac{\Gamma \vdash e{:}\tau_1 * \tau_2}{\Gamma \vdash \pi_r\ e{:}\tau_1}$$

$$\frac{\Gamma, z{:}\tau \vdash e{:}\tau'}{\Gamma \vdash \lambda z{:}\tau.e{:}\tau \to \tau'} \qquad \frac{\begin{array}{c}\Gamma \vdash e'{:}\tau \to \tau'\\ \Gamma \vdash e{:}\tau\end{array}}{\Gamma \vdash e'\ e{:}\tau'} \qquad \frac{\begin{array}{c}\Gamma \vdash e{:}\tau\\ \Gamma, z{:}\tau \vdash e'{:}\tau'\end{array}}{\Gamma \vdash \mathbf{let}\ z = e\ \ \mathbf{in}\ \ e'{:}\tau'}$$

$$\frac{\begin{array}{c}\Gamma, z{:}\tau \to \tau', x{:}\tau \vdash e{:}\tau'\\ \Gamma, z{:}\tau \to \tau' \vdash e'{:}\tau''\end{array}}{\Gamma \vdash \mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ e'{:}\tau''}$$

Figure 2.2: Lambda Calculi – Typing

2.2.1 Construct-time

The standard semantics, here called the *construct-time* or λ_c semantics, is recalled in Figure 2.1. We define a small-step reduction relation $e \longrightarrow e'$, using evaluation contexts E. Context composition and application are both written with a dot, e.g. $E.E'$ and $E.e$, instead of the usual heavier brackets $E[E'[-]]$ and $E[e]$. Standard capture-avoiding substitution of e for z in e' is written $\{e/z\}e'$.

At the bottom of Figure 2.1 the prediate e err is defined to identify stuck terms. An expression in λ_c is stuck if either (a) a projection of a non-pair value, or (b) an application of a non-function value, is in redex position.

We write $\mathrm{hb}(E)$, defined in Figure 2.3, for the list of binders around the hole of E.

2.2.2 Redex-time

The redex-time semantics is shown in Figure 2.4. Instead of substituting bindings of identifiers for values, as in the construct-time (app) and (let) rules, the λ_r semantics (and as we shall see later the λ_d semantics) introduces a **let** to record a binding of the abstraction's formal parameter to the application argument, e.g.

$$(\lambda z{:}\tau.e)u \quad \longrightarrow \quad \mathbf{let}\ z = u\ \ \mathbf{in}\ \ e$$

Define the list of hole-binders of E_3, written $\mathrm{hb}(E_3)$, by:

$$\begin{array}{lcl}
\mathrm{hb}(\_) & = & [] \\
\mathrm{hb}(E_3.A_1) & = & \mathrm{hb}(E_3) \\
\mathrm{hb}(E_3.(\mathbf{let}\ z{:}\tau = u\ \ \mathbf{in}\ \ \_)) & = & \mathrm{hb}(E_3), z \\
\mathrm{hb}(E_3.(\mathbf{letrec}\ z{:}\tau' = \lambda x_i{:}\tau.e\ \ \mathbf{in}\ \ \_)) & = & \mathrm{hb}(E_3), z
\end{array}$$

Figure 2.3: Auxiliary functions used in the definition of instantiation calculi

Redex-time λ_r

$$\begin{array}{llcl}
\text{Values} & u & ::= & n \mid () \mid (u, u') \mid \lambda z{:}\tau.e \mid \mathbf{let}\ z = u\ \ \mathbf{in}\ u' \mid \\
 & & & \mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ u \\
\text{Atomic evaluation contexts} & A_1 & ::= & (\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid u\ \_ \mid \\
 & & & \mathbf{let}\ z = \_\ \ \mathbf{in}\ \ e \\
\text{Atomic bind contexts} & A_2 & ::= & \mathbf{let}\ z = u\ \ \mathbf{in}\ \ \_ \mid \mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ \_ \\
\text{Evaluation contexts} & E_1 & ::= & \_ \mid E_1.A_1 \\
\text{Bind contexts} & E_2 & ::= & \_ \mid E_2.A_2 \\
\text{Reduction contexts} & E_3 & ::= & \_ \mid E_3.A_1 \mid E_3.A_2
\end{array}$$

Reduction Rules

(proj) $\pi_r(E_2.(u_1, u_2)) \longrightarrow E_2.u_r$

(app) $(E_2.(\lambda z{:}\tau.e))u \longrightarrow E_2.\mathbf{let}\ z = u\ \ \mathbf{in}\ \ e$ if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$

(inst) $\mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.z \longrightarrow \mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.u$

if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3)$

(instrec) $\mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ E_3.z \longrightarrow \mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ E_3.\lambda x{:}\tau.e$

if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(\lambda x{:}\tau.e) \notin \mathrm{hb}(E_3)$

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

Error Rules

Outermost-structure-manifest values $w ::= n \mid () \mid (u, u') \mid \lambda z{:}\tau.e$

(proj-err) $E_3.\pi_r(E_2.w)\mathsf{err}$ if $\neg\ \exists\ u_1, u_2.w = (u_1, u_2)$

(app-err) $E_3.(E_2.w)u\ \mathsf{err}$ if $\neg\ \exists(\lambda z{:}\tau.e).w = \lambda z{:}\tau.e$

Figure 2.4: Redex-time, instantiation-based Call-by-Value Lambda Calculus

This is reminiscent of an explicit substitution [ACCL90], save that here the **let** will not be percolated through the term structure, and also of the λ_{let}-calculus [AFM$^+$95], though we are in a CBV not CBN setting, and do not allow commutation of **let**s. In contrast, we must preserve let-binding structure, since our update primitive will depend on it.

Example (1) in Figure 2.5 illustrates (app), contrasting it with the substitution approach of the construct-time semantics. Note that the resulting $\mathbf{let}\ z = 8\ \ \mathbf{in}\ \ 7$ is a λ_r

	Construct-time λ_c	**Redex-time** λ_r	**Destruct-time** λ_d
(1)	$(\lambda z.7)8$	$(\lambda z.7)8$	$(\lambda z.7)8$
$\longrightarrow$	7	$\textbf{let}\ z = 8\ \textbf{in}\ 7$	$\textbf{let}\ z = 8\ \textbf{in}\ 7$
(2)	$\textbf{let}\ x = 5\ \textbf{in}\ \pi_1(x, x)$	$\textbf{let}\ x = 5\ \textbf{in}\ \pi_1(x, x)$	$\textbf{let}\ x = 5\ \textbf{in}\ \pi_1(x, x)$
$\longrightarrow$	$\pi_1(5, 5)$	$\textbf{let}\ x = 5\ \textbf{in}\ \pi_1(5, x)$	$\textbf{let}\ x = 5\ \textbf{in}\ x$
$\longrightarrow$	5	$\textbf{let}\ x = 5\ \textbf{in}\ \pi_1(5, 5)$	
$\longrightarrow$		$\textbf{let}\ x = 5\ \textbf{in}\ 5$	
(3)	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ \pi_1\ y$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ \pi_1\ y$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ \pi_1\ y$
$\longrightarrow$	$\textbf{let}\ y = (5, 6)\ \textbf{in}\ \pi_1\ y$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = (5, 6)\ \textbf{in}\ \pi_1\ y$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ \pi_1\ x$
$\longrightarrow$	$\pi_1(5, 6)$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = (5, 6)\ \textbf{in}\ \pi_1(5, 6)$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ \pi_1(5, 6)$
$\longrightarrow$	5	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = (5, 6)\ \textbf{in}\ 5$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ \textbf{let}\ y = x\ \textbf{in}\ 5$
(4)	$\pi_1(\pi_2(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (4, x))$	$\pi_1(\pi_2(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (4, x))$	$\pi_1(\pi_2(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (4, x))$
$\longrightarrow$	$\pi_1(\pi_2(4, (5, 6)))$	$\pi_1(\pi_2(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (4, (5, 6)))$	$\pi_1(\textbf{let}\ x = (5, 6)\ \textbf{in}\ x)$
$\longrightarrow$	$\pi_1(5, 6)$	$\pi_1(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (5, 6))$	$\pi_1(\textbf{let}\ x = (5, 6)\ \textbf{in}\ (5, 6))$
$\longrightarrow$	5	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ 5$	$\textbf{let}\ x = (5, 6)\ \textbf{in}\ 5$

Figure 2.5: Call-by-Value Lambda Calculi Examples

(and λ_d) value. Because values may involve **let**s, some clean-up is needed to extract the usual final result, for which we define

$$\begin{array}{rcll}
[\![n]\!]_{\mathsf{val}} & = & n & \\
[\![()]\!]_{\mathsf{val}} & = & () & \\
[\![(u, u')]\!]_{\mathsf{val}} & = & ([\![u]\!]_{\mathsf{val}}, [\![u']\!]_{\mathsf{val}}) & \\
[\![\lambda x{:}\tau.e]\!]_{\mathsf{val}} & = & \lambda x{:}\tau.e & \\
[\![\mathbf{let}\ z = u\ \ \mathbf{in}\ \ u']\!]_{\mathsf{val}} & = & \{[\![u]\!]_{\mathsf{val}}/z\}[\![u']\!]_{\mathsf{val}} & \\
[\![\mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ u]\!]_{\mathsf{val}} & = & \{\lambda x{:}\tau.\mathbf{letrec}\ z = \lambda x{:}\tau.e\ \ \mathbf{in}\ \ e/z\}[\![u]\!]_{\mathsf{val}} & \text{if } z \neq x \\
[\![z]\!]_{\mathsf{val}} & = & z &
\end{array}$$

which takes any value (λ_r or λ_d) and substitutes out the lets.

The semantics must allow reduction under lets: in addition to the atomic evaluation contexts A we had above (here A_1) we now have the binding contexts A_2 and reduction is closed under both. Redex-time variable instantiation is handled with the (inst) rule, which instantiates an occurrence of the identifier z in redex position with the innermost enclosing **let** that binds that identifier. The side-condition $z \notin \mathrm{hb}(E_3)$ ensures that the correct binding of z is used. Here $\mathrm{hb}(E)$ denotes the list of identifiers that bind around the hole of a context E, as defined in Figure 2.3. The other side-condition, $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3)$, which can always be achieved by alpha conversion, prevents identifier capture, making E_3 and **let** $z = u$ **in** _ transparent for u. Here fv(_) denotes the set of free identifiers of an expression or context.

Example (2) in Figure 2.5 illustrates identifier instantiation. While the construct-time strategy substitutes for x immediately, the redex-time strategy instantiates x under the **let**, following the evaluation order. Both this and the first example also illustrate a further aspect of the redex-time calculus: values u include let-bindings of the form **let** $z = u$ **in** u'. Intuitively, this is because a value should 'carry its bindings with it' preventing otherwise stuck applications, e.g. , $(\lambda x{:}\mathsf{int}.x)$(**let** $z = 3$ **in** $\lambda x{:}\mathsf{int}.z$). Note that identifiers are not values, so z, (z, z) and **let** $z = 3$ **in** (z, z) are not values. Values may contain free identifiers under lambdas, as usual, so $\lambda x{:}\mathsf{int}.z$ is an open value and **let** $z = 3$ **in** $\lambda x{:}\mathsf{int}.z$ is a closed value.

The (proj) and (app) rules are straightforward except for the additional binding context E_2. This is necessary as a value may now have some let bindings around a pair or lambda; terms such as π_1(**let** $z = 3$ **in** $(4, 5)$) or (more interestingly) π_1(**let** $z = 3$ **in** $(\lambda x{:}\mathsf{int}.z, 5)$) would otherwise be stuck. The purpose of the side condition for (app) is to prevent capture and can always be achieved by alpha conversion.

The bottom of Figure 2.4 defines the prediate e err that identifies terms stuck under λ_r. Its definition relies on the notion of *Outermost-structure-manifest* values. These are the values whose outer-most constructor is not a let or letrec binding. If such a value is about to be projected from and it is not a pair, or if it is about to be applied and it is not a function then the term is stuck; this is captured by the rules (proj-err) and (app-err).

2.2.3 Destruct-time

The redex-time strategy is appealingly simple, but it instantiates earlier than necessary. In Example (2) in Figure 2.5, both occurrences of x are instantiated before the projection reduction. However, we could delay resolving x until *after* the projection; we see this behaviour in the destruct-time semantics in the third column. In many dynamic rebinding scenarios it is desirable to instantiate as late as possible. For example, in dynamically updatable code we want to delay looking up a variable as long as possible, so as to acquire the most recent version.

To instantiate as late as possible, while remaining call-by-value, we only instantiate identifiers that are immediately under a projection or on the left-hand-side of an application. In these 'destruct' positions their values are about to be deconstructed, and so their outermost pair or lambda structure must be made manifest. The *destruct contexts* $R ::= \pi_r \_ \mid \_ \; u$ can be seen as the outer parts of the construct-time (proj) and (app) redexes. The choice of destruct contexts is determined by the basic redexes – for example, if we added arithmetic operations, we would need to instantiate identifiers of int type before using them.

The essential change from the redex-time semantics is that now any identifier is a value ($u ::= \ldots \mid z$). The (proj) and (app) rules are unchanged. The (inst) rule is replaced by two that together instantiate identifiers in destruct contexts R. The first (inst-1) copes with identifiers that are let-bound outside a destruct context, e.g. :

$$\textbf{let}\; z = (1,2)\;\textbf{in}\;\; \pi_1\; z \quad\longrightarrow\quad \textbf{let}\; z = (1,2)\;\textbf{in}\;\; \pi_1(1,2)$$

whereas in (inst-2) the let-binder and destruct context are the other way around:

$$\pi_1(\textbf{let}\; z = (1,2)\;\textbf{in}\;\; z) \quad\longrightarrow\quad \pi_1(\textbf{let}\; z = (1,2)\;\textbf{in}\; (1,2))$$

Further, we must be able to instantiate under nested bindings between the binding in question and its use. Therefore, (inst-2) must allow additional bindings E_2 and E_2' between R and the **let** and between the **let** and z. Similarly, (inst-1) must allow bindings

Destruct-time λ_d

Values	u	$::=$	$n \mid () \mid (u, u') \mid \lambda z{:}\tau.e \mid \textbf{let } z = u \textbf{ in } u' \mid$ $\textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } u \mid z$
Atomic evaluation contexts	A_1	$::=$	$(\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid u\ \_ \mid$ $\textbf{let } z = \_ \textbf{ in } e$
Atomic bind contexts	A_2	$::=$	$\textbf{let } z = u \textbf{ in } \_ \mid \textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } \_$
Evaluation contexts	E_1	$::=$	$\_ \mid E_1.A_1$
Bind contexts	E_2	$::=$	$\_ \mid E_2.A_2$
Reduction contexts	E_3	$::=$	$\_ \mid E_3.A_1 \mid E_3.A_2$
Destruct contexts	R	$::=$	$\pi_r\ \_ \mid \_\ u$

Reduction Rules

(proj) $\pi_r(E_2.(u_1, u_2)) \longrightarrow E_2.u_r$

(app) $(E_2.(\lambda z{:}\tau.e))u \longrightarrow E_2.\textbf{let } z = u \textbf{ in } e$ if $\text{fv}(u) \notin \text{hb}(E_2)$

(inst-1) $\textbf{let } z = u \textbf{ in } E_3.R.E_2.z \longrightarrow \textbf{let } z = u \textbf{ in } E_3.R.E_2.u$
if $z \notin \text{hb}(E_3, E_2)$ and $\text{fv}(u) \notin z, \text{hb}(E_3, E_2)$

(inst-2) $R.E_2.\textbf{let } z = u \textbf{ in } E_2'.z \longrightarrow R.E_2.\textbf{let } z = u \textbf{ in } E_2'.u$
if $z \notin \text{hb}(E_2')$ and $\text{fv}(u) \notin z, \text{hb}(E_2')$

(instrec-1)
$\textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } E_3.R.E_2.z \longrightarrow \textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } E_3.R.E_2.\lambda x{:}\tau.e$
if $z \notin \text{hb}(E_3, E_2)$ and $\text{fv}(\lambda x{:}\tau.e) \notin \text{hb}(E_3, E_2)$

(instrec-2)
$R.E_2.\textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } E_2'.z \longrightarrow R.E_2.\textbf{letrec } z = \lambda x{:}\tau.e \textbf{ in } E_2'.\lambda x{:}\tau.e$
if $z \notin \text{hb}(E_2')$ and $\text{fv}(\lambda x{:}\tau.e) \notin \text{hb}(E_2')$

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

Error Rules

Outermost-structure-manifest values $w ::= n \mid () \mid (u, u') \mid \lambda z{:}\tau.e \mid z$

(proj-err) $E_3.\pi_r(E_2.w)$err
if $\neg \exists\ u_1, u_2.w = (u_1, u_2)$ and $\neg \exists\ z \in \text{hb}(E_3, E_2).w = z$

(app-err) $E_3.(E_2.w)u$ err
$if \neg \exists (\lambda z{:}\tau.e).w = \lambda z{:}\tau.e and \neg \exists\ z \in \text{hb}(E_3, E_2).w = z$

Figure 2.6: Destruct-time, instantiation-based Call-by-Value Lambda Calculus

E_2 between the R and z; it must allow both binding and evaluation contexts E_3 between the **let** and the R, e.g. , for the instance

$$\begin{array}{ll} & \textbf{let } z = (1,(2,3)) \textbf{ in } \pi_1(\pi_2\ z) \\ \longrightarrow & \textbf{let } z = (1,(2,3)) \textbf{ in } \pi_1(\pi_2(1,(2,3))) \end{array}$$

with $E_3 = \pi_1$ _, $R = \pi_2$ _ and $E_2 =$ _. The conditions $z \notin \mathrm{hb}(E_3, E_2)$ and $z \notin \mathrm{hb}(E_2')$ ensure that the correct binding of z is used; the other conditions prevent capture and can always be achieved by alpha equivalence.

Example (3) illustrates a chain of instantiations, from outside-in for λ_r and from inside-out for λ_d.

The error rules for λ_d are the same as those for λ_r but with the extra condition that if the outermost-structure-manifest value is an identifier then it should not be in the hole binders of the surrounding context, for if this was the case then the term could proceed by an instantiation.

2.2.4 Properties

This subsection gives properties that sanity check our various λ-calculi: unique decomposition, preservation and safety. The proof of these facts is standard and can be found in the technical report [BHS$^+$03b]. Discussion of the more substantial result of observational equivalence is deferred to the next chapter.

First, we recall the important unique decomposition property of evaluation contexts for λ_c, essentially as in [FF87, p. 200], and generalise it to the more subtle evaluation contexts of λ_r and λ_d:

2.2.1 Theorem (Unique decomposition for λ_r and λ_d). *Let e be a closed expression. Then, in both the redex-time and destruct-time calculi, exactly one of the following holds:*

(i) e is a value;

(ii) e err;

(iii) there exists a triple (E_3, e', rn) such that $E_3.e' = e$ and e' is an instance of the left-hand side of rule rn.

Furthermore, if such a triple exists then it is unique. ❏

Note that the destruct-time error rules defining e err, given in Figure 2.6, must include cases for identifiers in destruct contexts that are not bound by enclosing **let**s and so are not instantiable, giving stuck non-value expressions. Determinacy is a trivial corollary. We also have type preservation and type safety properties for the three calculi.

2.2.2 Theorem (Type preservation for λ_c, λ_r and λ_d). *If $\Gamma \vdash e{:}\tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e'{:}\tau$.*

2.2.3 Theorem (Safety for λ_c, λ_r and λ_d). *If* $\vdash e{:}\tau$ *then* $\neg(e\ \mathit{err})$.

2.3 Update calculus

At the beginning of this chapter it was shown that the standard CBV λ-calculus did not meet our needs as the basis of a model for dynamic update. Using our new semantics for the lambda calculus based on delayed instantiation, we can now begin to explore models of dynamic update. The delayed instantiation will ensure that running code picks up any updated definitions as it executes and, furthermore, make it possible for expressions replaced at update time to make use of their surrounding environment by binding to the let-bound variables in scope.

Our aim in the design of this update mechanism is to allow as large a class of updates as possible. Thus, while our specification of update will be at the level of abstraction a programmer deals with, it will be fine grained so as not to limit potential updates. Two candidates exist as the unit of update in the λ-calculus, the most obvious unit being the function. However, while functions are a natural unit of abstraction in the λ-calculus, they do not serve as a good unit of updatability. This is mainly due to their anonymity, resulting in no natural way to refer to a specific function within a program, which poses problems when specifying what to update. A more appealing candidate is the let binding. Although **let**s are often a derived form of function, in the delayed instantiation calculi they play a key rôle. **let**s give a name to part of a computation and this name is programmer assigned, meaning that it often has some (informal) meaning to the programmer. So, accepting the let-binding as a unit of update leads us to consider an update as a set of pairs mapping let-bound names to expressions.

The reader familiar with programming languages, and the lambda calculus in particular, will know that it is useful to deal with binders *up to alpha conversion*, disregarding the exact name. Of course, in this setting, a let-bound name has no meaning from outside of the program – a fact most readily apparent if one thinks of binders as de Bruijn indices [dB72, dB80]. To preserve reasoning up to alpha equivalence, while having externally meaningful names, we use *tagged identifiers*. A tagged identifier, written x_k, consists of a constant name, x, together with an alpha-varying tag, k, which will range over integers for the purposes of examples. Thus $\lambda x_0.x_0$ is alpha equivalent to $\lambda x_1.x_1$ but not to $\lambda y_0.y_0$. By convention, math text x, y, z will be used for meta identifiers and sans serif $\mathsf{x}, \mathsf{y}, \mathsf{z}$ for concrete identifiers. As a further simplifying assumption we shall assume all identifiers in a program to be unique and we will omit the tag when it is superfluous to the discussion.

Simple Update Calculus: Syntax

Integers	n		Identifiers x, y, z	Tags i, j, k		
Types	τ	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid \tau * \tau' \mid \tau \to \tau'$			
Expressions	e	$::=$	$x_i \mid n \mid () \mid (e, e') \mid \pi_r\ e \mid \lambda x_i{:}\tau.e \mid e\ e' \mid \mathbf{let}\ z_k{:}\tau = e\ \ \mathbf{in}\ \ e' \mid$ $\mathbf{letrec}\ z_k{:}\tau = \lambda x_i{:}\tau.e\ \ \mathbf{in}\ \ e \mid \mathbf{update} \mid \mathbf{UpdEx}$			

Simple Update Calculus: Semantics

(upd-replace-ok)

$$\frac{S = \mathrm{rebind}(\mathrm{fv}(e), \mathrm{hb}(E_3)) \text{ is defined} \qquad \mathrm{env}(E_3) \vdash S(e){:}\tau \qquad \forall\, j.x_j \notin \mathrm{hb}(E_3')}{E_3.\mathbf{let}\ x_i{:}\tau = u\ \ \mathbf{in}\ \ E_3'.\mathbf{update} \xrightarrow{\{x \Leftarrow e\}} E_3.\mathbf{let}\ x_i{:}\tau = S(e)\ \mathbf{in}\ \ E_3'.()}$$

otherwise: $E_3.\mathbf{let}\ x_i{:}\tau = u\ \ \mathbf{in}\ \ E_3'.\mathbf{update} \xrightarrow{\{x \Leftarrow e\}} \mathbf{UpdEx}$

Figure 2.7: Simple Update Calculus: λ_{update}

2.3.1 Update mechanism

We now consider how to add a dynamic update facility to the late instantiation calculi developed in this chapter. We define the λ_{update}-calculus, given in Figure 2.7, as an extension of the λ_d calculus. In this calculus the programmer can place an **update** expression at points in the code where an update could occur; defining such updating 'safe points' is useful for ensuring programs behave properly [Hic01]. The intended semantics is that this expression will block, waiting for an update (possibly null) to be fed in. An update can modify any identifier that is within its scope (at update-time), for example in

$$\begin{array}{l}\mathbf{let}\ x_1 = (\mathbf{let}\ w_1 = 4\ \ \mathbf{in}\ \ w_1)\ \mathbf{in}\\ \mathbf{let}\ y_1 = \mathbf{update}\ \ \mathbf{in}\\ \mathbf{let}\ z_1 = 2\ \ \mathbf{in}\\ (x_1, z_1)\end{array}$$

x_1 may be modified by the update, but w_1, y_1 and z_1 may not. For simplicity we only allow a single identifier to be rebound to an expression of the same type, and we do not allow the introduction of new identifiers. Although both of these extensions would be straightforward, they do not add anything to the exposition.

We define the semantics of the update primitive using a labelled transition system, where the label is the updating expression. For example, supplying the label $\{x \Leftarrow \pi_1(3, 4)\}$ means that the nearest enclosing binding of x is replaced with a binding to $\pi_1(3, 4)$. Note that updates can be expressions, not just values – after an update the new expression, if not a value, will be in redex position. Further, they can be open, with free variables that become bound by the context of the **update**. To see why this is useful consider the following program:

let x = `input_int` **in**
let $y = x * x$ **in**
  update

Any expression given as an update to y will most likely require the use of x; the value of x was obtained via input at runtime and so is not known when writing an update. Thus, any expression that we replace $x * x$ with requires its free variables to be bound in the enclosing scope. We call this operation *rebinding*. Given a context E_3 and an expression e whose free variables are at most the hole binders of E_3 ($\mathrm{hb}(E_3)$), we define the rebinding of e in context E_3 to be the expression e with its free variables alpha-varied to match the closest enclosing identifier in E_3. More formally, we can define $\mathrm{rebind}(e, b)$ inductively on the structure of expressions, where b is the list of tagged identifiers binding around e:

$$\mathrm{rebind}(\Gamma, [])$$
$$\begin{cases} \text{undefined} & \text{if } \Gamma \text{ nonempty} \\ = \{\} & \text{otherwise} \end{cases}$$

$$\mathrm{rebind}(\Gamma, (L, (x_i{:}\tau)))$$
$$\begin{cases} \text{undefined, if } \exists\, j, \tau'.(x_j{:}\tau') \in\ \Gamma\ \wedge\ \tau' \neq \tau \\ = \{x_i/x_J\} \cup \mathrm{rebind}(\Gamma - x_J, L), \qquad \text{otherwise} \\ \quad \text{where } x_J = \{x_j \mid (x_j{:}\tau) \in\ \Gamma\} \end{cases}$$

(abusing notation to treat the partial function Γ as a set of tuples and writing $\{x_i/x_J\}$ for the substitution of x_i for all the $x_j\ \in\ x_J$).

Typing The static typing rule for **update** is trivial, as it is simply an expression of type unit. Naturally we have to perform some type checking at run-time; this is the second condition in the transition rule in Figure 2.7. Notice however, that we do not have to type-check the whole program; it suffices to check that the expression to be bound to the given identifier has the required type in the context that it will evaluate in. The other conditions of the transition rule are similarly straightforward. The first ensures that a rebinding substitution is defined, i.e. that the context E_3 has hole binders that are alpha-equivalent to the free variables of e. The third condition ensures that the binding being updated, x_i, is the closest such binding occurrence for x (notice that an equivalence class x is specified for the update, but that the closest enclosing member, x_i, of this class is chosen as the updated binding). These conditions are sufficient to ensure that the following theorems hold. We give only the cases involving the update keyword, as the rest are the same as for λ_d.

2.3.1 Theorem (Unique decomposition for λ_{update}).
Let e be a closed λ_{update} expression. Then exactly one of the following holds:

(i) e *is a value;*

(ii) e *err;*

(iii) there exists E_3, e', rn such that $E_3.e' = e$ and e' is an instance of the left-hand side of rule rn. Furthermore, if such a triple exists then it is unique, except for update, where we admit both an update reduction and an update failure.

Proof. As for the corresponding λ_d proof we generalise to open terms by proving that for all possibly open e exactly one of the following hold:

(i) e is a value;

(ii) e err;

(iii) there exists E_3, e', rn such that $E_3.e' = e$ and e' is an instance of the left-hand side of rule rn;

(iv) there exists E_3, R, E_2 such that $E_3.R.E_2.z = e$ and $z \notin \text{hb}(E_3.R.E_2)$ (unbound variable);

(v) there exists E_2 such that $E_2.z = e$ and $z \in \text{hb}(E_2)$ (bound variable).

The proof is by induction on the syntax of e, showing each case is unique as per the λ_d proof. The only extra construct to consider is **update**, in which (iii) holds where $E_3 = \_$, $e =$ **update** and e matches exactly the left-hand side of the update rule and the left-hand side of the update failure rule. ❑

2.3.2 Theorem (Type preservation for updates).
If $\vdash e{:}\tau$ *then either*

(i) if $e \longrightarrow e'$ *then* $\vdash e'{:}\tau$*; or*

(ii) if $e \xrightarrow{x \Leftarrow \hat{e}} e'$ *then* $\vdash e'{:}\tau$*; or*

(iii) $e =$ ***UpdEx***

Proof. The proof of the cases where (i) holds is the corresponding λ_d proof. The proof of (ii) is as follows. The only rule by which the reduction can follow is the update rule. Therefore

$$E_3.\mathbf{let}\ x_i{:}\tau = u\ \ \mathbf{in}\ \ E_3'.\mathbf{update} \xrightarrow{x \Leftarrow \hat{e}} E_3.\mathbf{let}\ x_i{:}\tau = S(\hat{e})\ \mathbf{in}\ \ E_3'.()$$

where

$$S = \mathrm{rebind}(\mathrm{fv}(e), \mathrm{hb}(E_3)) \tag{2.1}$$

$$\mathrm{env}(E_3) \vdash S(e){:}\tau \tag{2.2}$$

$$\forall\, j.\ x_j\ \notin \mathrm{hb}(E_3') \tag{2.3}$$

$$\vdash E_3.\mathbf{let}\ x_i{:}\tau = u\ \ \mathbf{in}\ \ E_3'.\mathbf{update}{:}\tau' \tag{2.4}$$

We are required to prove $\vdash\ E_3.\mathbf{let}\ x_i{:}\tau\ =\ S(\hat{e})\ \mathbf{in}\ \ E_3'.(){:}\tau'$. By E_3-inversion we have $\mathrm{env}(E_3)\ \vdash\ \mathbf{let}\ x_i{:}\tau\ =\ u\ \ \mathbf{in}\ \ E_3'.\mathbf{update}{:}\tau'$. By inversion of the typing relation $\mathrm{env}(E_3)\ \vdash\ u{:}\tau$ and $\mathrm{env}(E_3), x_i{:}\tau\ \vdash\ E_3'.\mathbf{update}{:}\tau'$ (*). By (*) and using E_3-inversion twice $\mathrm{env}(E_3), x_i{:}\tau\ \vdash\ E_3'.(){:}\tau'$. By 2.2 and let typing rule $\mathrm{env}(E_3)\ \vdash\ \mathbf{let}\ x_i{:}\tau\ =\ S(\hat{e})\ \mathbf{in}\ \ E_3'.(){:}\tau'$. ❑

2.3.3 Theorem (Safety for updates).
If $\vdash e{:}\tau$ *then* $\neg(e$ *err*$)$.

Proof. This follows as per the corresponding λ_d proof as **update** is not involved in the error rules. ❑

Higher-Order Functions Our use of delayed instantiation cleanly supports updating higher-order functions, a significant advance on previous treatments. Consider the following program:

let $f_1 = \lambda y_1.(\pi_2\ y_1, \pi_1\ y_1)$ **in**
let $w_1 = \lambda g_1.$**let** _ = **update in** $g_1(5,6)$ **in**
let $y_1 = f_1(3,4)$ **in**
let $z_1 = w_1\ f_1$ **in**
(y_1, z_1)

which contains an occurrence of **update** in the body of w_1. If, when w_1 is evaluated, we update the function f:

$$e \longrightarrow^{*} \xrightarrow{\{f \Leftarrow \lambda p_1.p_1\}} \longrightarrow^{*} u$$

we have $[\![u]\!]_{\mathsf{val}} = ((4,3),(5,6))$. Delayed instantiation plays a key role here: with the λ_c semantics, the result would be $[\![u]\!]_{\mathsf{val}} = ((4,3),(6,5))$; i.e. the update would not take effect because the g_1 in the body of w_1 would be substituted away by the (app) rule before the update occurs. Our semantics preserves both the structure of contexts and the names of variables so that updates can be expressed.

2.3.2 Conclusions

This chapter looked afresh at the semantics of the CBV lambda calculus from the point of view of dynamic binding and, in particular, dynamic update. We presented two alternative semantics, λ_r and λ_d, that differ from the usual semantics in two ways:

(1) they preserve the relationship between binder and bindee by using delayed instantiation rather than substitution as the mechanism for resolving variables; and

(2) they preserve the structure of the computation by not discarding or permuting environment bindings.

The difference between λ_r and λ_d is only when identifiers are instantiated. The former instantiates as soon as the variable comes into *redex* position, while the latter delays instantiation further, waiting until the identifier appears in a *destruct* position. Lastly we presented a simple calculus that supports dynamic update, providing evidence that delayed instantiation calculi are a suitable model of computation to use when studying dynamic update. In particular, an important part of update is the *rebinding* operation that binds the free variables of an expression to those in a given context, and this is straightforward to define and intuitive for delayed instantiation calculi.

3

λ_c,λ_r,λ_d Equivalence

It is not immediately apparent that the delayed instantiation reduction strategies presented in the previous chapter produce results that agree with the standard CBV strategy. Such a property is important if our claim that these calculi are suitable for reasoning about dynamic binding and dynamic update in a CBV lambda setting are to carry any weight. In this chapter we show that the reduction strategies of λ_r and λ_d are consistent, in a sense to be made clear, with that of the standard CBV λ-calculus. We start by reviewing the notion of equivalence in programming languages and sketching our proof strategy in §3.1. The proof of equivalence between λ_r and λ_c is then presented in detail in §3.2 followed by the equivalent proof between λ_d and λ_c in §3.3, presented in terms of where it differs from §3.2.

3.1 Equivalence

It is our intent in this chapter to prove the equivalence of reduction strategies. However, let us start out by considering equivalence of terms within a single language. A common notion of program equivalence is *contextual equivalence*, also known as *observational equivalence*. Two programs are contextually equivalent if they can be interchanged in some larger program without changing the final value. In sequential programming languages the notion of observation is usually the final result (or non-termination) or simply termination.

Based on this notion of program equivalence, if λ_c, λ_r and λ_d are equivalent reduction strategies then we expect every lambda term to produce the same result, irrespective of the strategy used. Moreover, we would also expect the associated contextual equivalence relations of the strategies to coincide. These are the facts we establish in this chapter. Formally, we will prove the following Theorem (where $[\![-]\!]_{\mathsf{val}}$ is the value-collapsing function defined in section 2.2.2)

3.1.1 Theorem (Observational Equivalence). *λ_c, λ_r and λ_d are all observationally equivalent at integer type:*

1. *If $\vdash e{:}\mathsf{int}$ and $e \longrightarrow_c^* n$ then for some u and u' we have $e \longrightarrow_r^* u$ and $e \longrightarrow_d^* u'$ and $[\![u]\!]_{\mathsf{val}} = [\![u']\!]_{\mathsf{val}} = n$.*
2. *If $\vdash e{:}\mathsf{int}$ and $e \longrightarrow_r^* u$ and $e \longrightarrow_d^* u'$ then for some n we have $e \longrightarrow_c^* n$ and $[\![u]\!]_{\mathsf{val}} = [\![u']\!]_{\mathsf{val}} = n$.*

We find that this theorem is a sufficient condition to ensure that the contextual equivalence relations coincide. To show this, we must first define exactly what we mean by contextual equivalence for each calculus. For λ_c this is standard:

3.1.2 Definition (Contextual Equivalence for λ_c). e and e' are *contextually equivalent* in λ_c, written $e \overset{\text{ctx}}{=}_c e'$ if and only if for all $\mathbb{C}$ such that $\vdash \mathbb{C}[e]{:}\mathsf{int}$ and $\vdash \mathbb{C}[e']{:}\mathsf{int}$ the following hold:

(i) if $\mathbb{C}[e] \longrightarrow_c^* n$ then $\mathbb{C}[e'] \longrightarrow_c^* n$

(ii) if $\mathbb{C}[e'] \longrightarrow_c^* n$ then $\mathbb{C}[e] \longrightarrow_c^* n$

❑

For the delayed instantiation calculi we define contextual equivalence to relate terms that reduce to values which collapse to identical terms under $[\![-]\!]_{\mathsf{val}}$. In other words the environment is substituted away before terms are compared at the end of the computation.

3.1.3 Definition (Contextual Equivalence for λ_r (λ_d)). e and e' are *contextually equivalent* in λ_r, written $e \overset{\text{ctx}}{=}_r e'$ if and only if for all $\mathbb{C}$ such that $\vdash \mathbb{C}[e]{:}\mathsf{int}$ and $\vdash \mathbb{C}[e']{:}\mathsf{int}$ the following hold:

(i) if $\mathbb{C}[e] \longrightarrow_r^* v$ then $\exists\, v'.\mathbb{C}[e'] \longrightarrow_r^* v' \wedge [\![v]\!]_{\mathsf{val}} = [\![v']\!]_{\mathsf{val}}$

(ii) if $\mathbb{C}[e'] \longrightarrow_r^* v$ then $\exists\, v'.\mathbb{C}[e] \longrightarrow_r^* v' \wedge [\![v]\!]_{\mathsf{val}} = [\![v']\!]_{\mathsf{val}}$

A similar definition, $e \overset{\text{ctx}}{=}_d e'$, holds for λ_d. ❑

The proof of the following theorem shows that coincidence of contextual equivalence follows from observational equivalence.

3.1.4 Theorem (Coincidence of Contextual Equivalence). *$\overset{\text{ctx}}{=}_c$, $\overset{\text{ctx}}{=}_r$ and $\overset{\text{ctx}}{=}_d$ are equivalent relations.*

Proof. It is sufficient to show for all e and e' that $e \overset{\text{ctx}}{=}_c e' \iff e \overset{\text{ctx}}{=}_r e'$ and $e \overset{\text{ctx}}{=}_c e' \iff e \overset{\text{ctx}}{=}_d e'$. We show just the former as the latter is similar.

➤*Case* $\Longrightarrow$: First prove point (i) in the definition of $\overset{\text{ctx}}{=}_r$. Suppose

$$e \overset{\text{ctx}}{=}_c e' \tag{3.1}$$

$$\vdash \mathbb{C}[e]{:}\mathsf{int} \tag{3.2}$$

$$\vdash \mathbb{C}[e']{:}\mathsf{int} \tag{3.3}$$

$$\mathbb{C}[e] \longrightarrow_r^* v \tag{3.4}$$

We prove $\exists\, v'.\mathbb{C}[e'] \longrightarrow_r^* v' \wedge [\![v]\!]_{\mathsf{val}} = [\![v']\!]_{\mathsf{val}}$. By 3.2, 3.4 and Observational Equivalence (Theorem 3.1.1) we have $\exists\, n.\mathbb{C}[e] \longrightarrow_c^* n \wedge n = [\![v]\!]_{\mathsf{val}}$. By 3.1 and previous fact $\mathbb{C}[e'] \longrightarrow_c^* n$. By 3.3, the previous fact and Observational Equivalence (Theorem 3.1.1) we have $\exists\, v''.\mathbb{C}[e'] \longrightarrow_c^* v'' \wedge n = [\![v'']\!]_{\mathsf{val}}$. It is immediate that $[\![v]\!]_{\mathsf{val}} = [\![v'']\!]_{\mathsf{val}}$, which together with the last fact proves the result.

Case (ii) is shown similarly.

➤*Case* $\Leftarrow$: Identical reasoning to the previous case. ❑

3.2 Observational Equivalence Between λ_r and λ_c

This section proves the λ_r part of Theorem 3.1.1, that is we prove Theorem 3.2.1. We do this by constructing a tight operational correspondence between the two calculi using the proof technique of bisimulation.

3.2.1 Theorem. *For all $e \in \lambda$ the following hold:*

1. $\vdash e{:}\mathsf{int} \Longrightarrow (e \longrightarrow_c^* n \Longrightarrow \exists\, v.\ e \longrightarrow_r^* u \wedge n = [\![v]\!]_{\mathsf{val}})$
2. $\vdash e{:}\mathsf{int} \Longrightarrow (e \longrightarrow_r^* v \Longrightarrow \exists\, n.\ e \longrightarrow_c^* n \wedge n = [\![v]\!]_{\mathsf{val}})$

It is well-known that statements like that in Theorem 3.2.1 do not admit a direct proof by induction. This is because the termination property of a term does not follow from that of its subterms and so a simple proof by structural induction does not suffice. For example, just because e and e' terminate at some value, does not necessarily imply that the application $e\ e'$ will terminate; consider $\lambda x.\Omega$ and 1, where Ω is a nonterminating computation. Logical relations and bisimulation are two proof techniques that provide a way to generalise the induction hypothesis to overcome this. We discuss both here as either is an equally viable technique, although we use the notion of bisimulation in our proof.

Logical relations A relation R is said to be logical if it has the property that whenever related functions f and f' are applied to related arguments e and e', the results $f\ e$ and $f'\ e'$ are related. These relations are defined by induction on the structure of types and are called logical because they respect the actions of the languages type constructors which by the curry-howard correspondence are related to the logical operators of intuitionistic logic. In particular, they respect the action of implication (function application) as stated above. While logical relations were first invented for denotational semantics, they have since been adapted to work directly on the syntax. If a logical relation can be defined for the property one wishes to prove, then a proof by induction can be carried out. The proof technique was first introduced by Tait [Tai67].

Bisimulation Bisimulation was originally developed for process calculi [Par81], but was later adapted to the lambda calculus and called *applicative bisimulation* by Abramsky [Abr90]. A relation R is a *simulation* w.r.t. some transition relation $\longrightarrow$ if whenever e_1 and e_2 are related and $e_1 \longrightarrow e_1'$ then $e_2 \longrightarrow e_2'$ for some e_2', such that the results e_1' and e_2' are related. If the statement obtained by interchanging e_1 with e_2 and e_1' with e_2' also holds, then it is a *bisimulation*.

While these techniques are described in terms of a single transition system, we require relations between transition systems, but the notions are readily extended in an obvious way.

The notion of simulation (resp. bisimulation) can be relaxed to what is known as a *weak simulation* (resp. weak bisimulation), we give a definition below following [Mil89].

3.2.2 Definition (Weak (Bi)Simulation). Given two transition systems $X \subseteq S_1 \times S_1$ and $Y \subseteq S_2 \times S_2$, we say that a relation $R \subseteq S_1 \times S_2$ relating states of X to states of Y is a *weak simulation from X to Y* if and only if for every $e_x\ R\ e_y$ the following holds:

$$e_x \longrightarrow_X e_x' \implies \exists\, e_y'.\ e_y \longrightarrow_Y^* e_y' \wedge e_x'\ R\ e_y'$$

If R^{-1} is a weak simulation from X to Y and a weak simulation from Y to X, then R is called a *weak bisimulation* between X and Y. ❑

For the purposes of proving our equivalence and obtaining a tight operational correspondence, we need a slight relaxation on this which we call an *Eventually Weak Simulation*:

3.2.3 Definition (Eventually Weak (Bi)Simulation). Given two transition systems $X \subseteq S_1 \times S_1$ and $Y \subseteq S_2 \times S_2$, we say that a relation $R \subseteq S_1 \times S_2$ relating states of X to states of Y is an *eventually weak simulation from X to Y* if and only if for every $e_x\ R\ e_y$ the following holds:

$$e_x \longrightarrow_X e'_x \implies \exists\, e'_y.\ e_y \longrightarrow^*_Y e'_y \ \wedge\ \exists\, n \geq 0.\ e'_x \rightarrow^n_X e''_x \ \wedge\ e''_x\ R\ e'_y$$

If the reverse implication holds it is a *eventually weak bisimulation between X and Y*. ❑

Observe that every bisimulation is a weak bisimulation and every weak bisimulation is an eventually weak bisimulation.

Informally, we require the weakness relaxation because λ_r performs more work than λ_c: while λ_c instantiates all instances of a bound variable in a single reduction step, λ_r requires reductions proportional to the number of occurrences of the variable. In addition, we require the eventually weak relaxation as λ_c's recursive function expansion results in $\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \ \mathbf{in}\ \ e$ while in λ_r we obtain $\lambda x.e$ instead. Upon application of these functions λ_c requires an extra reduction when compared to λ_r, i.e.:

$$\begin{aligned}(\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \ \mathbf{in}\ \ e)v \ &\longrightarrow_c\ \mathbf{letrec}\ z = \lambda x.e\ \ \mathbf{in}\ \{v/x\}e \\ &\longrightarrow_c\ \{\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \ \mathbf{in}\ \ e/z\}\{v/x\}e\end{aligned}$$

but,

$$(\lambda x.e)u \ \longrightarrow_r\ \mathbf{let}\ x = u\ \ \mathbf{in}\ \ e$$

A similar situation holds for λ_d.

3.2.1 An Overview

Before we embark on the detailed proof we provide a high-level overview. The proof goes like this:

1. Try to explicitly construct an eventually weak bisimulation relation (henceforth EWB relation) between λ_r and λ_c terms; fail.

2. Introduce an annotated calculus, λ'_r, that preserves more information during reduction so that the EWB we seek may be defined.

3. Introduce normal forms. Eventually weak simulations only guarantee that two terms in the relation eventually reduce to another two terms in the relation. In order to produce a proof we must identify criteria for the terms to be in the relation. This leads us to introduce the notion of Instantiation and Zero Normal Forms.

4. Establish some basic properties of our definitions.

5. Show that the relation we defined *is* an EWB.

6. Show that the termination relation for λ'_r and λ_c coincide. Sadly it will not be the case that our relation (R) relates values only to values. Therefore we show that whenever eRv or vRe' then e and e' terminate.

7. Show that λ_r and λ_c are observationally equivalent. This follows from the existence of the EWB and the fact that the termination relations coincide.

The same process is repeated for λ_d in section 3.3.

3.2.2 An Eventually Weak Bisimulation Relation

In this section we define a candidate eventually weak bisimulation relation R and establish its basic properties. In later sections we prove that this relation is actually an EWB between λ_c and λ_r. We have already seen in section 2.2.2 a function $[\![-]\!]_{\mathsf{val}}$ that transforms values built by instantiation reduction into the corresponding value λ_c reduction would have built. It is reasonable, although incorrect as we shall see, to seek to define R by lifting $[\![-]\!]_{\mathsf{val}}$ to act on expressions (call it $[\![-]\!]$) and defining R as the set $\{(e, e') \mid e' = [\![e]\!]\}$. Let us define such a function and see why it fails to be an EWB. We lift to expressions by acting inductively on applications, projections and on let expressions that bind non-values. On let expressions that bind values we act the same as $[\![-]\!]_{\mathsf{val}}$. Our presentation differs from that used to define $[\![-]\!]_{\mathsf{val}}$ in that instead of using substitutions to remove let bindings we instead use an environment, Φ, that records a mapping from variables to λ_c terms as we find it convenient for later developments. The (erroneous) definition of $[\![-]\!]^{\Phi}$ is as follows

$$\begin{array}{lcl}
[\![z]\!]^{\Phi} & = & \Phi(z) \\
[\![n]\!]^{\Phi} & = & n \\
[\![(e_1, e_2)]\!]^{\Phi} & = & ([\![e_1]\!]^{\Phi}, [\![e_2]\!]^{\Phi}) \\
[\![\pi_r\ e]\!]^{\Phi} & = & \pi_r [\![e]\!]^{\Phi} \\
[\![\lambda x.e]\!]^{\Phi} & = & \lambda x.[\![e]\!]^{\Phi\ ,\ x \mapsto x} \\
[\![e_1\ e_2]\!]^{\Phi} & = & [\![e_1]\!]^{\Phi} [\![e_2]\!]^{\Phi} \\
[\![\textbf{let}\ z = v\ \ \textbf{in}\ \ e]\!]^{\Phi} & = & [\![e]\!]^{\Phi\ ,\ z \mapsto [\![v]\!]^{\Phi}} \\
[\![\textbf{let}\ z = e_1\ \ \textbf{in}\ \ e_2]\!]^{\Phi} & = & \textbf{let}\ z = [\![e_1]\!]^{\Phi}\ \textbf{in}\ [\![e_2]\!]^{\Phi\ ,\ z \mapsto z} \\
[\![\textbf{letrec}\ z = \lambda x.e_1\ \ \textbf{in}\ \ e_2]\!]^{\Phi} & = & [\![e_2]\!]^{\Phi\ ,\ z \mapsto [\![\mu(z,x,e_1)]\!]^{\Phi}}
\end{array}$$

In the last part we make use of shorthand defined below.

3.2.4 Definition (Recursive abbreviation)**.** We write $\mu(z, x, \tau, e)$ for $\lambda x{:}\tau.\textbf{letrec}\ z = \lambda x{:}\tau.e\ \ \textbf{in}\ \ e$ and overload it for annotated terms (defined in Figure 3.1) such that $\mu(z, x, \tau, a)$ stands for $\lambda x{:}\tau.\textbf{letrec}_1\ \ z = \lambda x{:}\tau.a\ \ \textbf{in}\ \ a$. Whenever the types are clear from the context we write $\mu(z, x, e)$ and $\mu(z, x, a)$ respectively. ❑

While the function $[\![-]\!]^{-}$ converts r-expressions to c-expressions it has two serious problems stemming from the fact that it identifies "the environment" in an erroneous way:

(i) Not every value-binding let is part of "the environment". Let bindings on the outside of a computation that bind values are morally part of the computation's environment; their values are used by the computation, but the terms they bind are fully computed. For example, in

$$\textbf{let}\ x = 5\ \ \textbf{in}\ \ \textbf{let}\ y = 6\ \ \textbf{in}\ \ \textbf{let}\ z = \pi_1(x, y)\ \textbf{in}\ \ z$$

the values bound to x and y are used by the computation under them, but no more computation occurs above or within them. The variables x and y are part of the environment, but the **let** binding z is in the part of the program that is yet to be computed: it is part of the computation. The problem with our definition of $[\![-]\!]^{-}$ is that it identifies a let binding as part of the environment if it binds a value, a condition that is too weak. To see this consider

$$\begin{array}{ll}
 & [\![\textbf{let}\ x = 5\ \ \textbf{in}\ \ \textbf{let}\ y = \pi_1(1\ ,\ 2)\ \textbf{in}\ \ \textbf{let}\ z = 3\ \ \textbf{in}\ (x\ ,(y\ ,\ z))]\!]^{\Phi} \\
= & \textbf{let}\ y = \pi_1(1,2)\ \textbf{in}\ (y, (3, 5))
\end{array}$$

Here the inner-most **let** binding a value is substituted away in advance of the time λ_c reduction would substitute it away, changing the evaluation order. The problem here is that the third let which binds z is morally part of the program – a *program-let* – and should not be substituted away. On the other hand, the **let** binding x is morally part of the environment – an *environment-let* – and is correctly substituted away.

(ii) The recursive unrolling rules for the two calculi produce structurally different results. For example, the term **letrec** $z = \lambda x.e$ **in** z can do an instantiation of z under λ_r to become

$$\textbf{letrec}\ z = \lambda x.e\ \ \textbf{in}\ \lambda x.e \tag{3.5}$$

and the corresponding λ_c term according to $[\![-]\!]^-$ is

$$[\![\mu(z, x, e)]\!]^{\varnothing} \tag{3.6}$$

which is a value. Unfortunately, the resulting terms are not related under our (erroneous) R, that is, applying $[\![-]\!]^-$ to 3.5 does not yield 3.6. In λ_c the environment containing the recursive binding of z is carried with the recursive function, whereas in λ_r (and λ_d) the recursive binding is not duplicated.

To make a correct definition of $[\![-]\!]^-$ we must formally recognise the separation of what is "the program" and what is "the environment" in a given λ_r term. To do this we introduce an intermediate language, $\lambda_{r'}$, given in Figures 3.1 and 3.2, that distinguishes the two forms of **let**: $\textbf{let}_0$ for environment-**let**s and $\textbf{let}_1$ for program-**let**s. The (zero) and (zerorec) reductions convert a 1-tagged **let**/**letrec** into a 0-tagged **let**/**letrec** whenever a 1-tagged **let**/**letrec** binding a value is in redex position. These reductions correspond to substituting **let**s away in λ_c.

Similarly, a tagging scheme is employed for distinguishing between functions and recursive unrollings of functions. Whenever a variable bound by a **letrec** is instantiated we tag the function with the name of the **letrec** it came from, e.g. z will be instantiated to $\lambda^z x.a$. See the (instrec) rule in Figure 3.2.

Notation We say that $\lambda^z x.a$ is a *recursive function* and call z in that term a *recursive variable*. Write $\mathrm{frv}(a)$ (the *free recursive variables* in a) for the recursive variables in a not bound by an enclosing **letrec** and $\mathrm{frf}(a)$ (the *free recursive functions* in a) for the recursive functions whose recursive variables are in $\mathrm{frv}(a)$.

Notation Whenever we want to specify that a reduction is of a specific type we will label the transition with its name, e.g. $a \xrightarrow{\text{inst}} r' a'$. As a generalisation of this, we will write $a \xrightarrow{\text{insts}} r' a'$ to mean that a can do an inst or instrec transition to become a', and we call the action an *inst reduction*. Similarly, we write $a \xrightarrow{\text{zeros}} r' a'$ for a zero or zerorec transition and call it a *zero reduction*.

Intuitively, environment-**let**s are only found on the outside of a computation and subsequent to a program-**let** occurring there should be no more occurrences of environment-**let**s. Also, we expect tagged functions to only be introduced by the computation and not by the user, therefore we do not expect to find tagged functions under lambdas or below program **let**s. For these reasons, we work only with well-formed annotated terms defined as follows.

3.2.5 Definition (Well-formedness)**.** We write $\mathrm{wf}[a]$ to denote that a term a is well-formed, in the sense of the definition below. The definition uses an auxiliary predicate $\mathrm{noenv}(a)$ which asserts that a nor any of its subexpressions contain environment syntax ($\mathbf{let}_0$,$\mathbf{letrec}_0$ or λ^z).

$$\begin{array}{lcl}
\mathrm{wf}[z] & \iff & \mathrm{t} \\
\mathrm{wf}[n] & \iff & \mathrm{t} \\
\mathrm{wf}[()] & \iff & \mathrm{t} \\
\mathrm{wf}[(a, a')] & \iff & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \\
\mathrm{wf}[\pi_r\ a] & \iff & \mathrm{wf}[a] \\
\mathrm{wf}[\lambda^j x{:}\tau.a] & \iff & \mathrm{wf}[a] \wedge \mathrm{noenv}(a) \\
\mathrm{wf}[a\ a'] & \iff & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \\
\mathrm{wf}[\mathbf{let}_0\ z = a\ \mathbf{in}\ a'] & \iff & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \wedge a\ \mathsf{val} \\
\mathrm{wf}[\mathbf{let}_1\ z = a\ \mathbf{in}\ a'] & \iff & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \wedge \mathrm{noenv}(a') \\
\mathrm{wf}[\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ a'] & \iff & \mathrm{wf}[\lambda x{:}\tau.a] \wedge \mathrm{wf}[a'] \\
\mathrm{wf}[\mathbf{letrec}_1\ z = \lambda x{:}\tau.a\ \mathbf{in}\ a'] & \iff & \mathrm{wf}[\lambda x{:}\tau.a] \wedge \mathrm{wf}[a'] \wedge \mathrm{noenv}(a')
\end{array}$$

We extend $\mathrm{wf}[-]$ to act on A_1 contexts by including $\mathrm{wf}[\_] = \mathrm{t}$ and otherwise remaining unchanged from its action on expressions. On reduction contexts we define it as follows:

$$\begin{array}{lcl}
\mathrm{wf}[\_] & \iff & \mathrm{t} \\
\mathrm{wf}[\_.E_3] & \iff & \mathrm{wf}[E_3] \\
\mathrm{wf}[A_1.E_3] & \iff & \mathrm{wf}[A_1] \wedge \mathrm{wf}[E_3] \\
\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3] & \iff & \mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ \_] \wedge \mathrm{wf}[E_3] \\
\mathrm{wf}[\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ E_3] & \iff & \mathrm{wf}[\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ \_] \wedge \mathrm{wf}[E_3]
\end{array}$$

Integers	n		
Identifiers	x, y, z		
Types	τ	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid \tau * \tau' \mid \tau \to \tau'$
Exprs	a	$::=$	$z \mid n \mid () \mid (a, a') \mid \pi_r\ a \mid \lambda^j x.a$
		$\mid$	$a\,a' \mid \mathbf{let}_m\ \ z = a\ \ \mathbf{in}\ \ a' \mid \mathbf{letrec}_m\ \ z = \lambda x{:}\tau.a\ \ \mathbf{in}\ \ a'$
Annotations	m	$::=$	$0 \mid 1$
	j	$::=$	$\cdot \mid z$
	r	$::=$	$1 \mid 2$

Figure 3.1: Annotated syntax for $\lambda_{r'}$

❑

The correct definition of $[\![-]\!]^-$ is given in Figure 3.3, which relies on an environment Φ whose definition is given below.

3.2.6 Definition (Environment)**.** An *environment* Φ is a list containing pairs whose first component is an identifier and whose second component is a c-value or an identifier. An environment is well-formed if the following hold:

(i) Whenever $(x, z) \in \Phi$ then $x = z$.

(ii) Whenever $(x, e) \in \Phi$ then forall $z \in \text{fv}(e)$ it holds that $z \leq_\Phi x$ where $\leq_\Phi$ is the ordering of the identifiers in Φ.

(iii) All of the first components of the pairs in the list are distinct.

When Φ is well-formed we write $\Phi \blacktriangleleft$ [1]. We write $\Phi, z \mapsto v$ for the disjoint extension of Φ forming a new environment and $\Phi[z \mapsto v]$ for the environment acting as Φ, but mapping z to v. ❑

Note that in the above definition if Φ is well-formed it does not necessarily follow that the extensions are well formed. Clause (i) is a simplifying assumption reflecting the fact that if an environment maps an identifier to a non-value, then it maps it to itself. Clause (ii) is a closure property ensuring that variables which occur free in the environment have definitions further up the environment. Clause (iii) ensures each identifier is defined only once, allowing us to treat an environment as a finite partial function without ambiguity.

[1] We adopt this strange notation as later we extend environment well-formedness to environment well-formedness w.r.t. a term a, which we write $\Phi \blacktriangleleft a$

Reduction contexts

Values	u	$::=$	$n \mid () \mid (u,u') \mid \lambda x{:}\tau.a \mid \mathbf{let}_0\ z{:}\tau = u\ \mathbf{in}\ u$
		$\mid$	$\mathbf{letrec}_0\ z{:}\tau = \lambda x{:}\tau.a\ \mathbf{in}\ u$
Atomic eval ctxts	A_1	$::=$	$(\_,a) \mid (u,\_) \mid \pi_r\ \_ \mid \_\ a \mid \lambda x{:}\tau.a\ \_$
		$\mid$	$\mathbf{let}_1\ z{:}\tau = \_\ \mathbf{in}\ a$
Atomic bind ctxs	A_2	$::=$	$\mathbf{let}_0\ z{:}\tau = u\ \mathbf{in}\ \_ \mid \mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ \_$
Eval ctxts	E_1	$::=$	$\_ \mid E_1.A_1$
Bind ctxts	E_2	$::=$	$\_ \mid E_2.A_2$
Reduction ctxts	E_3	$::=$	$\_ \mid E_3.A_1 \mid E_3.A_2$

Reduction rules

(proj)	$\pi_r(E_2.(u_1,u_2))$	$\longrightarrow$	$E_2.u_r$
(app)	$(E_2.(\lambda x{:}\tau.a)u)$ if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$	$\longrightarrow$	$E_2.\mathbf{let}_0\ x = u\ \mathbf{in}\ a$
(inst)	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.z$ if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(u) \notin z,\mathrm{hb}(E_3)$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.u$
(instrec)	$\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ E_3.z$ if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(\lambda x{:}\tau.a) \notin z,\mathrm{hb}(E_3)$	$\longrightarrow$	$\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ E_3.\lambda^z x{:}\tau.a$
(zero)	$\mathbf{let}_1\ z = u\ \mathbf{in}\ a$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ a$
(zerorec)	$\mathbf{letrec}_1\ z = \lambda x{:}\tau.a\ \mathbf{in}\ a'$	$\longrightarrow$	$\mathbf{letrec}_0\ z = \lambda x{:}\tau.a\ \mathbf{in}\ a'$
(cong)	$\dfrac{a \longrightarrow a'}{E_3.a \longrightarrow E_3.a'}$		

Figure 3.2: $\lambda_{r'}$ calculus

The function $[\![-]\!]^\Phi$ is not well defined for all terms. Given a well-formed environment Φ the function $[\![-]\!]^\Phi$ on lambda terms acts on variables by looking them up in the environment Φ. Thus it is well defined only for terms whose free variables are contained in the domain of Φ. Additionally, because recursive functions, $\lambda^z x.a$, mention a variable z, whenever we apply $[\![-]\!]^\Phi$ to such a term z should be mapped by Φ. In this case the environment and the term both associate a function with z and we must ensure that the terms they associate with it are compatible. The correct definition of compatible is that the body of the function in the environment is the image of the one in the term under $[\![-]\!]^{\Phi'}$ where Φ' is the bindings above z in Φ extended to map the free variables x and z to themselves. The following definition formalises this compatibility of environment and term.

Here we introduce a function that expresses the correspondence between well-formed $\lambda_{r'}$ terms built through instantiation and λ terms built through substitution (in the sense of λ_c reduction). $[\![\, a \,]\!]^{\Phi}$ is a function mapping a $\lambda_{r'}$ expression a and an environment Φ to a λ_c expression. We note that in each of the cases where we extend the environment to associate an identifier with a value, we can ensure that the identifier is fresh for the environment by alpha conversion.

$$\begin{array}{rcll}
[\![\, z \,]\!]^{\Phi} & \triangleq & \Phi(z) & \\
[\![\, n \,]\!]^{\Phi} & \triangleq & n & \\
[\![\, () \,]\!]^{\Phi} & \triangleq & () & \\
[\![\, (a, a') \,]\!]^{\Phi} & \triangleq & ([\![\, a \,]\!]^{\Phi}, [\![\, a' \,]\!]^{\Phi}) & \\
[\![\, \pi_r\ a \,]\!]^{\Phi} & \triangleq & \pi_r [\![\, a \,]\!]^{\Phi} & \\
[\![\, \lambda x{:}\tau.a \,]\!]^{\Phi} & \triangleq & \lambda x{:}\tau.[\![\, a \,]\!]^{\Phi\,,\, x \mapsto x} & x \notin \mathrm{dom}(\Phi) \\
[\![\, \lambda^{z} x{:}\tau.a \,]\!]^{\Phi} & \triangleq & \Phi(z) & \\
[\![\, a\, a' \,]\!]^{\Phi} & \triangleq & [\![\, a \,]\!]^{\Phi}\, [\![\, a' \,]\!]^{\Phi} & \\
[\![\, \mathbf{let}_0\ \ z = a\ \ \mathbf{in}\ \ a' \,]\!]^{\Phi} & \triangleq & [\![\, a' \,]\!]^{\Phi\,,\, z \mapsto [\![\, a \,]\!]^{\Phi}} & z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{let}_1\ \ z = a\ \ \mathbf{in}\ \ a' \,]\!]^{\Phi} & \triangleq & \mathbf{let}\ z = [\![\, a \,]\!]^{\Phi}\ \mathbf{in}\ [\![\, a' \,]\!]^{\Phi\,,\, z \mapsto z} & z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{letrec}_0\ \ z = \lambda x.a\ \ \mathbf{in}\ \ a' \,]\!]^{\Phi} & \triangleq & [\![\, a' \,]\!]^{\Phi\,,\, z \mapsto [\![\, \mu(z,x,a) \,]\!]^{\Phi}} & z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{letrec}_1\ \ z = \lambda x.a\ \ \mathbf{in}\ \ a' \,]\!]^{\Phi} & \triangleq & \mathbf{letrec}\ z = [\![\, \lambda x.a \,]\!]^{\Phi\,,\, z \mapsto z}\ \mathbf{in}\ [\![\, a' \,]\!]^{\Phi\,,\, z \mapsto z} & \\
 & & & z \notin \mathrm{dom}(\Phi)
\end{array}$$

We extend $[\![\, - \,]\!]^{-}$ to act on A_1 contexts by adding the clause $[\![\, \_ \,]\!]^{\Phi} = \_$. On reduction contexts we define the action as:

$$\begin{array}{rcl}
[\![\, \mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ \_.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, E_3 \,]\!]^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}} \\
[\![\, \mathbf{letrec}_0\ \ z = \lambda x.a\ \ \mathbf{in}\ \ \_.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, E_3 \,]\!]^{\Phi\,,\, z \mapsto [\![\, \mu(z,x,a) \,]\!]^{\Phi}} \\
[\![\, A_1.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, A_1 \,]\!]^{\Phi}.[\![\, E_3 \,]\!]^{\Phi} \\
[\![\, \_.E_3 \,]\!]^{\Phi} & \triangleq & \_.[\![\, E_3 \,]\!]^{\Phi}
\end{array}$$

Figure 3.3: Instantiate-substitute correspondence and its extension to evaluation contexts

3.2.7 Definition (Environment-Term Compatibility). A term a is compatible with an environment Φ, written $\Phi \blacktriangleleft a$, if and only if the following hold

(i) the environment is well-formed: $\Phi \blacktriangleleft$

(ii) $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi)$

(iii) for all $\lambda^z x.\hat{a} \ \in \mathrm{frf}(a)$ there exists Φ_1, Φ_2, e such that $\Phi = \Phi_1, z \mapsto \lambda x.\mathbf{letrec}\ z = \lambda x.e\ \ \mathbf{in}\ \ e, \Phi_2$ and $\Phi_1, x \mapsto x, z \mapsto z \blacktriangleleft \hat{a}$ and $e = [\![\, \hat{a} \,]\!]^{\Phi_1\ ,\ x \mapsto x\ ,\ z \mapsto z}$.

The definition extends naturally to evaluation contexts $\Phi \blacktriangleleft E_3$. ❑

Environment-term compatibility is closed under reduction, as the following lemma proves.

3.2.8 Lemma ($- \blacktriangleleft -$ is Closed Under Reduction). $\Phi \blacktriangleleft a\ \wedge\ a\ \longrightarrow_{r'}\ a' \implies \Phi \blacktriangleleft a'$

Proof. Proof is by induction on the transition relation derivation. For each case the three points in the definition of $\Phi \blacktriangleleft a'$ must be established. Point (i) is immediate from assumptions. Point (ii) can be established easily by showing the property $\mathrm{fv}(a) \subseteq \mathrm{fv}(a')$. Finally, to show point (iii) observe that for all $\lambda^z x.\hat{a} \ \in \mathrm{frf}(a)$ the condition required holds by assumption, therefore it suffices to show $\mathrm{frf}(a) \subseteq \mathrm{frf}(a')$, which can be established by an induction on the transition relation. ❑

Defining R

We have now defined a function to relate the intermediate language $\lambda_{r'}$ to λ_c. However, we wish to define a candidate bisimulation relation, R, between λ_r and λ_c. We therefore need a way of relating unannotated λ_r terms and annotated $\lambda_{r'}$ ones. Fortunately this is straightforward. Going from unannotated to annotated it is assumed that the whole term is part of the program, so all **let**s are 1-annotated and functions are left unannotated, while the reverse direction is the forgetful function that removes all annotations. The former is called *inject* and the latter *erase*. Their definitions are given in Figure 3.2.9 and 3.2.10 respectively.

3.2.9 Definition (inject). $\iota[\_]$ is a function that converts unannotated λ terms to annotated $\lambda_{r'}$ terms.

$$\begin{array}{lcl}
\iota[z] & = & z \\
\iota[n] & = & n \\
\iota[()] & = & () \\
\iota[\pi_r\ e] & = & \pi_r\iota[e] \\
\iota[(e, e')] & = & (\iota[e], \iota[e']) \\
\iota[\lambda x{:}\tau.e] & = & \lambda x{:}\tau.\iota[e] \\
\iota[e\ e'] & = & \iota[e]\iota[e'] \\
\iota[\textbf{let}\ z = e\ \ \textbf{in}\ \ e'] & = & \textbf{let}_1\ \ z = \iota[e]\ \textbf{in}\ \iota[e'] \\
\iota[\textbf{letrec}\ z =_\lambda x.e\ \ \textbf{in}\ \ e'] & = & \textbf{letrec}_1\ \ z = \iota[\lambda x.e]\ \textbf{in}\ \iota[e']
\end{array}$$

❑

3.2.10 Definition (erase). $\epsilon[\_]$ is a function that converts annotated $\lambda_{r'}$ terms to unannotated λ terms.

$$\begin{array}{lcl}
\epsilon[z] & = & z \\
\epsilon[n] & = & n \\
\epsilon[()] & = & () \\
\epsilon[\pi_r\ a] & = & \pi_r\epsilon[a] \\
\epsilon[(a, a')] & = & (\epsilon[a], \epsilon[a']) \\
\epsilon[\lambda x{:}\tau.a] & = & \lambda x{:}\tau.\epsilon[a] \\
\epsilon[\lambda^z x{:}\tau.a] & = & \lambda x{:}\tau.\epsilon[a] \\
\epsilon[a\ a'] & = & \epsilon[a]\,\epsilon[a'] \\
\epsilon[\textbf{let}_0\ \ z = a\ \ \textbf{in}\ \ a'] & = & \textbf{let}\ z = \epsilon[a]\ \textbf{in}\ \epsilon[a'] \\
\epsilon[\textbf{let}_1\ \ z = a\ \ \textbf{in}\ \ a'] & = & \textbf{let}\ z = \epsilon[a]\ \textbf{in}\ \epsilon[a'] \\
\epsilon[\textbf{letrec}_0\ \ z =_\lambda x.a\ \ \textbf{in}\ \ a'] & = & \textbf{let}\ z = \epsilon[\lambda x.a]\ \textbf{in}\ \epsilon[a'] \\
\epsilon[\textbf{letrec}_1\ \ z =_\lambda x.a\ \ \textbf{in}\ \ a'] & = & \textbf{let}\ z = \epsilon[\lambda x.a]\ \textbf{in}\ \epsilon[a']
\end{array}$$

We extend $\epsilon[-]$ to act on A_1 contexts by adding the clause $\epsilon[\_] = \_$. On reduction contexts we define the action as:

$$\begin{array}{rcl}
\epsilon[\mathbf{let}_0\ \ z = a\ \ \mathbf{in}\ \ \_.E_3] & = & \mathbf{let}\ z = \epsilon[a]\ \mathbf{in}\ \epsilon[E_3] \\
\epsilon[\mathbf{letrec}_0\ \ z = \lambda x.a\ \ \mathbf{in}\ \ \_.E_3] & = & \mathbf{letrec}_0\ \ z = \epsilon[\lambda x.a]\ \mathbf{in}\ \epsilon[E_3] \\
\epsilon[A_1.E_3] & = & \epsilon[A_1].\epsilon[E_3] \\
\epsilon[\_.E_3] & = & \_.\epsilon[E_3]
\end{array}$$

❑

We can now give the resulting definition of the candidate weak bisimulation R.

3.2.11 Definition (Candidate Eventually Weak Bisimulation).

$$R \equiv \{(e, e') \mid \exists\ a.\ \mathrm{wf}[a]\ \wedge\ a\ \ \text{closed}\ \wedge\ e = [\![\ a\ ]\!]^{\varnothing}\ \wedge\ e' = \epsilon[a]\}$$

❑

The relation is on unannotated terms, but defined in terms of projections out of an annotated $\lambda_{r'}$ term; $[\![\ -\ ]\!]^{-}$ forms the corresponding λ_c term and $\epsilon[-]$ the corresponding λ_r term.

The goal is to show that if we start with identical terms then the two reduction systems reduce them to equivalent values, we therefore need identical terms to be related by R. We check this sanity property.

3.2.12 Definition (id_λ). id_λ is the identity relation on closed lambda terms: $\mathrm{id}_\lambda = \{(e, e) \mid e\ \in \lambda\ \wedge\ e\ \ \text{closed}\}$ ❑

3.2.13 Lemma (R Contains Identity). *The candidate bisimulation R contains id_λ.*

Proof. It suffices to prove $\epsilon[\iota[e]] = e$ and $[\![\ \iota[e]\ ]\!]^{\varnothing} = e$. The first is clear from the definitions. The second can be proved by induction on e. ❑

Basic Properties of Constituents of R

This section establishes some basic properties of $[\![\ -\ ]\!]^{-}$, $\epsilon[-]$ and $\mathrm{wf}[-]$ – the basic building blocks of R – as well as environment well-formedness conditions. We are mainly interested in how the operations distribute over our syntax, that they preserve values and that well-formedness of terms is preserved by reduction.

The following definition provides the link between $\lambda_{r'}$ evaluation contexts and environments Φ.

3.2.14 Definition (Binding Context). $\mathcal{E}_c[E_3]^\Phi$ builds an environment corresponding to the binding context of the $\lambda_{r'}$ reduction context E_3 using the environment Φ.

$$\begin{array}{lcl}
\mathcal{E}_c[\_]^\Phi & = & \varnothing \\
\mathcal{E}_c[\_.E_3]^\Phi & = & \mathcal{E}_c[E_3]^\Phi \\
\mathcal{E}_c[A_1.E_3]^\Phi & = & \mathcal{E}_c[E_3]^\Phi \\
\mathcal{E}_c[\textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ \_.E_3]^\Phi & = & z \mapsto [\![\, u \,]\!]^\Phi, \mathcal{E}_c[E_3]^{\Phi\ ,\ z \mapsto [\![\, u \,]\!]^\Phi} \\
\mathcal{E}_c[\textbf{letrec}_0\ \ z = \lambda x.a\ \ \textbf{in}\ \ \_.E_3]^\Phi & = & z \mapsto [\![\, \mu(z,x,a) \,]\!]^\Phi, \mathcal{E}_c[E_3]^{\Phi\ ,\ z \mapsto [\![\, \mu(z,x,a) \,]\!]^\Phi}
\end{array}$$

The context E_3 and the environment Φ must be compatible in the sense that $\mathrm{fv}(E_3) \subseteq \mathrm{dom}(\Phi)$ and $\mathrm{hb}(E_3)$ must be unique. ❑

When extending an environment with a value care must be taken to ensure the resulting environment is well-formed. The following facts are useful in doing this.

3.2.15 Proposition (Environment Properties).

(i) If $\Phi \blacktriangleleft u$ *and* $z \notin dom(\Phi)$ *then* $\Phi, z \mapsto [\![\, u \,]\!]^\Phi \blacktriangleleft$

(ii) If $\Phi \blacktriangleleft a$ *and* $\Phi, \Phi' \blacktriangleleft$ *then* $\Phi, \Phi' \blacktriangleleft a$

(iii) If $\Phi \blacktriangleleft E_3.a$ *then* $\Phi, \mathcal{E}_c[E_3]^\Phi \blacktriangleleft a$

❑

We can extend parts (ii) and (iii) of the previous lemma to contexts to conclude the following.

3.2.16 Corollary (Environment Context Properties).

(i) If $\Phi \blacktriangleleft E_3$ *and* $\Phi, \Phi' \blacktriangleleft$ *then* $\Phi, \Phi' \blacktriangleleft E_3$

(ii) If $\Phi \blacktriangleleft E_3.E_3'$ *then* $\Phi, \mathcal{E}_c[E_3]^\Phi \blacktriangleleft E_3'$

❑

3.2.17 Lemma ($[\![-]\!]^-$ Value Preservation). $\Phi \blacktriangleleft u \ \wedge\ wf[u] \implies [\![\, u \,]\!]^\Phi$ *cval*

Proof. We prove by induction on u. $[\![-]\!]^\Phi$ clearly preserves n and $()$, so these cases are trivial. The pair case follows by application of IH. In the function case functions in $\lambda_{r'}$

are transformed to functions in λ_c, and functions are values. This leaves the 0-tagged-**let**/**letrec** case (we show just the case for **let**):

➤*Case* $\textbf{let}_0 \ = u_1 \ \textbf{in} \ u_2$ **:** Assume $\Phi \blacktriangleleft \textbf{let}_0 \ z = u_1 \ \textbf{in} \ u_2$ and $\mathrm{wf}[\textbf{let}_0 \ z = u_1 \ \textbf{in} \ u_2]$. Note $[\![\textbf{let}_0 \ z = u_1 \ \textbf{in} \ u_2]\!]^{\Phi} = [\![u_2]\!]^{\Phi'}$ where $\Phi' = \Phi, z \mapsto [\![u_1]\!]^{\Phi}$. By Environment Properties (iii) (Proposition 3.2.15) $\Phi' \blacktriangleleft u_2$ and by definition of $\mathrm{wf}[-]$, $\mathrm{wf}[u_2]$. By induction $[\![u_2]\!]^{\Phi'}$ cval as required. ❑

3.2.18 Lemma (Well-Formed Context Decomposition). *$wf[E_3.a] \iff wf[E_3] \wedge wf[a]$*

Proof. ($\Rightarrow$) Assume $\mathrm{wf}[E_3.a]$ and note that $\mathrm{wf}[-]$ acts on contexts in the same way it acts on expressions, thus $\mathrm{wf}[E_3]$. Furthermore having a surrounding context can only impose stricter conditions upon a, thus $\mathrm{wf}[a]$.

($\Leftarrow$) Assume $\mathrm{wf}[E_3] \wedge \mathrm{wf}[a]$ and note that $\mathrm{wf}[-]$ can only fail if the $\mathrm{noenv}(-)$ or value checks fail. No holes in E_3 coincide with these checks, thus $\mathrm{wf}[E_3.a]$. ❑

3.2.19 Lemma ($\lambda_{r'}$ reduction preserves well-formedness). *$wf[a] \wedge a \longrightarrow_{r'} a' \implies wf[a']$*

Proof. The proof proceeds by an easy induction on the transition relation for $\lambda_{r'}$ using the definition of $\mathrm{wf}[-]$ and Well-Formed Context Decomposition (Lemma 3.2.18) to prove each case. ❑

We now prove some conditions under which a change of environment in $[\![-]\!]^{-}$ leaves the image unchanged.

3.2.20 Proposition ($[\![-]\!]^{-}$ Environment Properties).

(i) *If $wf[a]$ and $\Phi, x \mapsto x \blacktriangleleft a$ and $\Phi, x \mapsto v, \Phi' \blacktriangleleft$ then $\{v/x\}[\![a]\!]^{\Phi\,,\,x\mapsto x\,,\,\Phi'} = [\![a]\!]^{\Phi\,,\,x\mapsto v\,,\,\Phi'}$*

(ii) *If $\Phi \blacktriangleleft a$ and $\Phi, \Phi' \blacktriangleleft a$ then $[\![a]\!]^{\Phi} = [\![a]\!]^{\Phi\,,\,\Phi'}$*

(iii) *If $\Phi_1, \Phi_2, \Phi_3, \Phi_4 \blacktriangleleft a$ and $\Phi_1, \Phi_3, \Phi_2, \Phi_4 \blacktriangleleft a$ then $[\![a]\!]^{\Phi_1\,,\,\Phi_2\,,\,\Phi_3\,,\,\Phi_4} = [\![a]\!]^{\Phi_1\,,\,\Phi_3\,,\,\Phi_2\,,\,\Phi_4}$.*

Proof. First prove (i) by induction on a. Cases () and n are trivial.

➤*Case* z **:** Let $\Phi_x \equiv \Phi, x \mapsto x, \Phi'$ and $\Phi_v \equiv \Phi, x \mapsto v, \Phi'$. Assume $\Phi, x \mapsto x \blacktriangleleft z$ and $\Phi_x \blacktriangleleft$ and $\mathrm{wf}[z]$. As we know that $\Phi, x \mapsto x \blacktriangleleft z$ then $[\Phi, x \mapsto x](z) = z$ or $[\Phi, x \mapsto x](z) = v'$ for some c-value v'. Let us consider each case:

➤*Case* $[\Phi, x \mapsto x](z) = z$ **:** If $z = x$ holds then

$$\{v/x\}[\![z]\!]^{\Phi_x} = \{v/x\}[\Phi_x](z) = \{v/x\}x = v = [\![z]\!]^{\Phi_v}$$

if not then

$$\{v/x\}[\![z]\!]^{\Phi_x} = \{v/x\}[\Phi_x](z) = \{v/x\}z = z = [\![z]\!]^{\Phi_v}$$

➤*Case* $[\Phi, x \mapsto x](z) = v'$ **:** In this case

$$\{v/x\}[\![z]\!]^{\Phi_x} = \{v/x\}[\Phi_x](z) = \{v/x\}v'$$

holds and it suffices to show that $\{v/x\}v' = v' = [\![z]\!]^{\Phi_v}$. The second equality is true by assumption. To show the first equality is suffices to prove $x \notin \mathrm{fv}(v')$. As $\Phi, x \mapsto x \blacktriangleleft z$ and $\Phi, x \mapsto x, \Phi'$ is well-formed, then $z \notin \mathrm{dom}(\Phi')$. Thus $z \in \mathrm{dom}(\Phi)$. By environment well-formedness $\Phi(z)$ can only contain free variables defined before it in the environment, and x is defined after it.

➤*Case* $\lambda^{\cdot} z{:}\tau.a$ **:** Assume $\Phi \blacktriangleleft \lambda z{:}\tau.a$ and $\mathrm{wf}[\lambda z{:}\tau.a]$. First note that by alpha conversion $z \notin \{x\} \cup \mathrm{dom}(\Phi)$ can be ensured. Then $\Phi\ ,\ x \mapsto x \blacktriangleleft a$ and $\mathrm{wf}[a]$, so by induction

$$\{v/x\}[\![a]\!]^{\Phi\ ,\ x \mapsto x} = [\![a]\!]^{\Phi\ ,\ x \mapsto v}$$

from which the result follows by lambda abstracting on z.

➤*Case* $\lambda^z x.a$ **:** Let $\Phi_x \equiv \Phi, x \mapsto x, \Phi'$ and $\Phi_v \equiv \Phi, x \mapsto v, \Phi'$. Assume $\Phi, x \mapsto x \blacktriangleleft \lambda^z x.a \ \wedge\ \Phi_x \blacktriangleleft v \ \wedge\ \mathrm{wf}[\lambda^z x.a]$. $x \neq z$ by well-formedness (z is a free recursive variable and so must map to a function) and $z \notin \mathrm{dom}(\Phi')$ as $\Phi, x \mapsto x \blacktriangleleft \lambda^z x.a$. Therefore $\Phi_x(z) = \Phi_v(z) = \Phi(z) = [\![\lambda^z x.a]\!]^{\Phi_x} = [\![\lambda^z x.a]\!]^{\Phi_v}$ and it suffices to show that $x \notin \mathrm{fv}(\Phi(z))$. This holds as $\Phi, x \mapsto x \blacktriangleleft \lambda^z x.a$.

The rest of the cases follow a similar pattern.

Part (ii) is clear from the definition of $[\![-]\!]^{-}$ and well-formed environments.

Part (iii). We proceed by induction on the structure of a. The base terms $()$ and n are trivially true. The interesting case is variables, but here we note that both environments agree on all variable assignments. All other cases follow by applying the induction hypothesis. (Observe that all bound identifiers in a can be alpha converted not to clash with those in $\mathrm{dom}(\Phi_1\ ,\ \Phi_2\ ,\ \Phi_3\ ,\ \Phi_4) = \mathrm{dom}(\Phi_1\ ,\ \Phi_3\ ,\ \Phi_2\ ,\ \Phi_4)$). ❑

3.2.21 Lemma ($\llbracket - \rrbracket$ Outer Value Preservation). *For all $\lambda_{r'}$ values u:*

(a) If $\mathrm{wf}[u]$, $\Phi \blacktriangleleft u$ and $\llbracket u \rrbracket^{\Phi} = \lambda x{:}\tau.e$ then there exists E_2, a, j such that $u = E_2.\lambda^j x{:}\tau.a$

(b) $\llbracket u \rrbracket^{\Phi} = (v_1, v_2) \implies \exists\ E_2, u_1, u_2.\ u = E_2.(u_1, u_2)$

Proof. We prove (a) by induction on u. The cases of $n, (), (u_1, u_2)$ are trivially true as $\llbracket - \rrbracket^{-}$ on these terms can not result in a term of the form $\lambda x{:}\tau.e$. The case $\lambda^j x{:}\tau.a$ results in a function when $\llbracket - \rrbracket^{-}$ is applied, but it is already of the right form if one chooses $E_2 = \_$. This leaves the (let) and (letrec) cases:

➤*Case* $\textbf{let}_0\ \ z = u_1\ \ \textbf{in}\ \ u_2$ **:** Assume $\mathrm{wf}[\textbf{let}_0\ \ z = u_1\ \ \textbf{in}\ \ u_2]$; $\Phi \blacktriangleleft \textbf{let}_0\ \ z = u_1\ \ \textbf{in}\ \ u_2$ and $\llbracket \textbf{let}_0\ z = u_1\ \textbf{in}\ u_2 \rrbracket^{\Phi} = \lambda x{:}\tau.e$ (*). By definition of well-formedness $\mathrm{wf}[u_1]\ \wedge\ \mathrm{wf}[u_2]$. It is easy to show that $\Phi \blacktriangleleft u_1$ and $\Phi\ ,\ z \mapsto \llbracket u_1 \rrbracket^{\Phi} \blacktriangleleft u_2$ follows by Environment Properties (Proposition 3.2.15) . By definition of $\llbracket - \rrbracket^{-}$ and (*) $\llbracket \textbf{let}_0\ z = u_1\ \textbf{in}\ u_2 \rrbracket^{\Phi} = \llbracket u_2 \rrbracket^{\Phi\ ,\ z \mapsto \llbracket u_1 \rrbracket^{\Phi}} = \lambda x{:}\tau.e$, thus by induction there exists E_2', a', j' such that $u_2 = E_2'.\lambda^{j'} x{:}\tau.a'$. The result follows by choosing $E_2 = (\textbf{let}_0\ \ z = u_1\ \ \textbf{in}\ \ \_.E_2'), a = a'$ and $j = j'$.

➤*Case* $\textbf{letrec}_0\ \ z = \lambda x{:}\tau.a\ \ \textbf{in}\ \ u$ **:** By the definition of $\llbracket - \rrbracket^{-}$ and environment well formedness there is an e such that

$$\llbracket \textbf{letrec}_0\ \ z = \lambda x{:}\tau.a\ \ \textbf{in}\ \ u \rrbracket^{\Phi} = \llbracket u \rrbracket^{\Phi\ ,\ z \mapsto \llbracket \mu(z,x,a) \rrbracket^{\Phi}} = \lambda x{:}\tau.e$$

from which the result follows by induction.

(b) is proved by a similar induction on u. ❑

3.2.22 Lemma ($\llbracket - \rrbracket^{-}$ Distribution Over Contexts). *For all E_3, Φ and a, if $\Phi \blacktriangleleft E_3.a$ and $\mathrm{wf}[E_3.a]$ then $\llbracket E_3.a \rrbracket^{\Phi} = \llbracket E_3 \rrbracket^{\Phi}.\llbracket a \rrbracket^{\Phi\ ,\mathcal{E}_c[E_3]^{\Phi}}$*

Proof. We prove by induction on E_3. We elide the $\textbf{letrec}_0$ context case as it is similar to the $\textbf{let}_0$ case.

➤*Case* $A_1.E_3'$ **:**

$$\begin{array}{rcll}
\llbracket A_1.E_3'.a \rrbracket^{\Phi} & = & \llbracket A_1 \rrbracket^{\Phi}.\ \llbracket E_3'.a \rrbracket^{\Phi} & \\
& = & \llbracket A_1 \rrbracket^{\Phi}.\ \llbracket E_3' \rrbracket^{\Phi}.\ \llbracket a \rrbracket^{\Phi\ ,\mathcal{E}_c[E_3']^{\Phi}} & (*) \\
& = & \llbracket A_1.E_3' \rrbracket^{\Phi}.\ \llbracket a \rrbracket^{\Phi\ ,\mathcal{E}_c[A_1.E_3']^{\Phi}} &
\end{array}$$

By Well-Formed Context Decomposition (Lemma 3.2.18) we have $\mathrm{wf}[E_3'.a]$, and by Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft E_3'.a$, thus by induction (*) holds.

➤*Case* $\mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3'$:

$$\begin{aligned}
[\![\, \mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3'.a \,]\!]^{\Phi} &= [\![\, E_3'.a \,]\!]^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}} \\
&= [\![\, E_3' \,]\!]^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}} .\, [\![\, a \,]\!]^{\Phi'} & (*) \\
&\qquad \text{where } \Phi' = \Phi\ ,\ z \mapsto [\![\, u \,]\!]^{\Phi}, \mathcal{E}_c[E_3']^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}} \\
&= [\![\, \mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3' \,]\!]^{\Phi} .\, [\![\, a \,]\!]^{\Phi\,, \mathcal{E}_c[\mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3']^{\Phi}} & (**)
\end{aligned}$$

By definition of $\mathrm{wf}[-]$ we have $\mathrm{wf}[E_3'.a]$, and by Environment Properties (iii) (Proposition 3.2.15) $\Phi' \blacktriangleleft E_3'.a$, thus by induction (*) holds. By definition of $\mathcal{E}_c[-]^{-}$, (**) is equivalent to (*).

❑

3.2.23 Lemma ($[\![\, - \,]\!]$ Preserves Contexts). *If* $\Phi \blacktriangleleft E_3$ *and* $\mathit{wf}[E_3]$ *then there exists a* λ_c *reduction context* E *such that* $[\![\, E_3 \,]\!]^{\Phi} = E$.

Proof. We proceed by induction on the structure of E_3:

➤*Case* $\_$: $[\![\, \_ \,]\!]^{\Phi} = \_$ which is a valid λ_c reduction context.

➤*Case* $A_1.E_3'$: Assume $\Phi \blacktriangleleft A_1.E_3'$ (3.7) and $\mathrm{wf}[A_1.E_3']$ (3.8). From the definition of $[\![\, - \,]\!]$ on contexts (Figure 3.3) $[\![\, A_1.E_3' \,]\!]^{\Phi} = [\![\, A_1 \,]\!]^{\Phi}.[\![\, E_3' \,]\!]^{\Phi}$. By Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft E_3'$ and by Well-Formed Context Decomposition (Lemma 3.2.18) $\mathrm{wf}[E_3']$. From these derived facts and induction there exists an E such that $E = [\![\, E_3' \,]\!]^{\Phi}$. We are left to show that $[\![\, A_1 \,]\!]^{\Phi}$ is a valid λ_c reduction context for every A_1:

➤*Case* $(\_, a)$: Follows directly from definition

➤*Case* $(u, \_)$: $[\![\, (u, \_) \,]\!]^{\Phi} = ([\![\, u \,]\!]^{\Phi}, \_)$ which is a λ_c context only if $[\![\, u \,]\!]^{\Phi}$ cval. From 3.7 we can deduce $\Phi \blacktriangleleft u$. From 3.8 we conclude $\mathrm{wf}[u]$. By these last two facts and $[\![\, - \,]\!]^{-}$ Value Preservation (Lemma 3.2.17) $[\![\, u \,]\!]^{\Phi}$ cval as required.

The rest of the A_1 cases are similar to one of the above two.

➤*Case* $\mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3'$: Assume $\Phi \blacktriangleleft \mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3'$ and $\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3']$. From the definition of $[\![\, - \,]\!]$ on contexts (Figure 3.3) $[\![\, \mathbf{let}_0\ z = u\ \mathbf{in}\ E_3' \,]\!]^{\Phi} = [\![\, E_3' \,]\!]^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}}$. By Environment Properties (iii) (Proposition 3.2.15) $z \mapsto [\![\, u \,]\!]\Phi$ and $\Phi \blacktriangleleft E_3'$, and by Well-Formed Context Decomposition (Lemma 3.2.18) $\mathrm{wf}[E_3']$. Thus by induction there exists an E such that $[\![\, E_3 \,]\!]^{\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}} = E$.

❑

We now establish a similar set of properties for $\epsilon[-]$, although the definition is considerably simpler making the proofs routine.

3.2.24 Lemma ($\epsilon[-]$ Value Preservation)**.** $wf[u] \implies \epsilon[u]$ *rval* ❑

3.2.25 Lemma ($\epsilon[-]$ Distributes Over Contexts)**.** $\epsilon[E_3.a] = \epsilon[E_3].\epsilon[a]$ ❑

3.2.26 Lemma ($\epsilon[-]$ Preserves Contexts)**.** *If* $wf[E_3]$ *then there exists a* λ_r *reduction context* E_3' *such that* $\epsilon[E_3] = E_3'$.

Proof. By induction on E_3. We elide the **let**$_0$ and **letrec**$_0$ cases as they are similar to the A_1 case.

➤*Case* _ **:** trivial

➤*Case* $A_1.E_3$ **:** Assume that $\mathrm{wf}[A_1.E_3]$. By Well-Formed Context Decomposition (Lemma 3.2.18) $\mathrm{wf}[E_3]$. By induction there exists a λ_r context E_3' such that $\epsilon[E_3] = E_3'$. Now $\epsilon[A_1.E_3] = \epsilon[A_1].\epsilon[E_3] = \epsilon[A_1].E_3'$ and furthermore, by $\epsilon[-]$ Value Preservation (Lemma 3.2.24) it is easy to verify that for each A_1, $\epsilon[A_1]$ is a valid λ_r atomic context.

➤*Case* **let**$_0$ $z = u$ **in** $\_.E_3$ **:** Similar to the previous case.

❑

3.2.27 Lemma ($\epsilon[-]$ Outer Value Preservation)**.** *For all* $\lambda_{r'}$ *values* u*:*

(a) *If* $\mathrm{wf}[u]$ *and* $\epsilon[u] = E_2.\lambda x{:}\tau.e$ *then there exists* $\hat{E}_2, a, z, j$ *such that* $u = \hat{E}_2.\lambda^j x{:}\tau.a$

(b) $\epsilon[u] = E_2.(v_1, v_2) \implies \exists\, \hat{E}_2, u_1, u_2.\ u = \hat{E}_2.(u_1, u_2)$

❑

3.2.3 R is an Eventually Weak Bisimulation

In this section we show that R, as defined in Definition 3.2.11, is a weak bisimulation between λ_c and λ_r. To do this we factor the problem into two simulations, one from λ_c to λ_r and the other in the reverse direction. These simulations are further factored through the annotated calculus $\lambda_{r'}$. This process is shown for each simulation in Figures 3.4 and 3.5. Proving that R is a bisimulation amounts to showing that these two diagrams hold for all a. The diagrams are intended to be read left-to-right and top-down and show how reductions are related between the various calculi. The upper and lower quadrants of each diagram are proved separately.

$$\begin{tikzcd}
c & e \arrow[rr, "l", "\frac{1}{2}"' near end, "c"' end] & & e' \\
r' & a \arrow[u, dotted, "{[\![-]\!]^\Phi}"] \arrow[d, dotted, "{\epsilon[-]}"'] \arrow[r, "insts", "*\ r'"'] & \bullet \arrow[ul, dotted, "{[\![-]\!]^\Phi}"'] \arrow[d, dotted, "{\epsilon[-]}"] \arrow[r, "\hat{l}", "r'"'] & a' \arrow[u, dotted, "{[\![-]\!]^\Phi}"'] \arrow[d, dotted, "{\epsilon[-]}"] \\
r & \hat{e} \arrow[r, "insts", "*\ r"'] & \bullet \arrow[r, "l", "\text{if } l \neq \text{zeros}"'] \arrow[r, equal, bend right, "\text{if } l = \text{zeros}"'] & \hat{e}'
\end{tikzcd}$$

Figure 3.4: Operational reasoning of cr-simulation

$$\begin{tikzcd}
r & \hat{e} \arrow[rr, "l", "r"' end] & & \hat{e}' \\
r' & a \arrow[u, dotted, "{\epsilon[-]}"] \arrow[d, dotted, "{[\![-]\!]^\Phi}"'] \arrow[r, "zeros", "*\ r'"'] & \bullet \arrow[ul, dotted, "{\epsilon[-]}"'] \arrow[d, dotted, "{[\![-]\!]^\Phi}"'] \arrow[r, "\check{l}", "r'"'] & a' \arrow[u, dotted, "{\epsilon[-]}"'] \arrow[d, dotted, "{[\![-]\!]^\Phi}", "\frac{1}{2}"'] \\
c & e \arrow[r, "lets", "*\ c"'] & \bullet \arrow[r, "l", "\text{if } l \neq \text{insts}"'] \arrow[r, equal, bend right, "\text{if } l = \text{insts}"'] & e'
\end{tikzcd}$$

Figure 3.5: Operational reasoning of rc-simulation

An Eventually Weak CR-Simulation

We first consider the upper quadrant of the cr-simulation diagram, which can be read as follows. Suppose $e\ R\ \hat{e}$ via a (i.e. a satisfies the existential in the definition of R), and e reduces in one step to e' by a λ_c reduction, call it l, then a can perform a finite sequence of instantiation reductions followed by a non-instantiation reduction $\hat{l}$. This reduction matches l and results in a' such that $[\![\ a'\ ]\!]^\Phi = e'$.

The argument presupposes that instantiation reductions to a leave the image of a under $[\![-]\!]^\Phi$ unchanged. The following lemma confirms this.

3.2.28 Lemma ($[\![-]\!]^-$ Invariant Under Insts). $wf[a] \;\wedge\; \Phi \blacktriangleleft a \;\wedge\; a \xrightarrow{insts}{}^{*}_{r'} a' \implies [\![a]\!]^{\Phi} = [\![a']\!]^{\Phi}$

Proof. We first prove the single reduction case by induction on $a \xrightarrow{\text{insts}}_{r'} a'$. Every case is trivial except (inst), (instrec) and (cong):

➤*Case* (inst) : Assume $\text{wf}[\textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ E_3.z]$ and $\Phi \blacktriangleleft \textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ E_3.z$. We are required to prove that applying $[\![-]\!]^{\Phi}$ to the left and right hand side of this rule results in the same term. First take the LHS:

$$\begin{aligned}
[\![\textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ E_3.z]\!]^{\Phi} &= [\![E_3.z]\!]^{\Phi\,,\, z \mapsto [\![u]\!]^{\Phi}} \\
&= [\![z]\!]^{\Phi\,,\,\Phi'} && (3.9) \\
&\text{where } \Phi' = z \mapsto [\![u]\!]^{\Phi}, \mathcal{E}_c[E_3]^{\Phi\,,\, z \mapsto [\![u]\!]^{\Phi}} \\
&= [\Phi, \Phi'](z) && (3.10) \\
&= [\![u]\!]^{\Phi}
\end{aligned}$$

3.9 follows from $[\![-]\!]^-$ Distribution Over Contexts (Lemma 3.2.22) and 3.10 follows as $z \notin \text{hb}(E_3)$ by the side condition of rule. Now take the RHS:

$$[\![\textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ E_3.u]\!]^{\Phi} = [\![u]\!]^{\Phi\,,\,\Phi'} \tag{3.11}$$

We are left to show that $[\![u]\!]^{\Phi} = [\![u]\!]^{\Phi\,,\,\Phi'}$.

By side condition of the (inst) reduction rule $\text{fv}(u) \notin \text{hb}(E_3)$ and by alpha conversion $z \notin \text{fv}(u)$. It follows that $\text{fv}(u) \notin \text{dom}(\Phi')$. By Environment Properties (iii) (Proposition 3.2.15) $\Phi' \blacktriangleleft u$, thus by $[\![-]\!]^-$ Environment Properties (ii) (Proposition 3.2.20) $[\![u]\!]^{\Phi} = [\![u]\!]^{\Phi\,,\,\Phi'}$, as required.

➤*Case* (instrec) : Follows directly from the definition of $[\![-]\!]^-$ and $[\![-]\!]^-$ Distribution Over Contexts (Lemma 3.2.22) .

➤*Case* (cong) : Assume $\text{wf}[E_3.a]$ and $\Phi \blacktriangleleft E_3.a$. By Well-Formed Context Decomposition (Lemma 3.2.18) $\text{wf}[a]$. Let $\Phi' = \Phi, \mathcal{E}_c[E_3]^{\Phi}$, then by Environment Properties (Proposition 3.2.15) $\Phi' \blacktriangleleft a$. By induction $[\![a]\!]^{\Phi'} = [\![a']\!]^{\Phi'}$ (*). Now $[\![E_3.a]\!]^{\Phi} = [\![a]\!]^{\Phi'}$ and $[\![E_3.a']\!]^{\Phi} = [\![a']\!]^{\Phi'}$ by Environment Properties (iii) (Proposition 3.2.15) , thus by (*) we are done.

The multiple step case follows by induction on the number of reductions. ❑

The validity of the upper quadrant also relies on the fact that every contiguous sequence of instantiations is finite. That is, we eventually reach a term that cannot reduce via an instantiation. We say such a term is in instantiation normal form, which is formally defined in Definition 3.2.29 and extended to open terms in Definition 3.2.30. The obvious variant of these definitions hold for λ_r as well.

3.2.29 Definition (INF). A term a is in *instantiation normal form* (INF) if and only if there does not exist an a' such that $a \xrightarrow{\text{insts}} a'$. We write $a\ \mathsf{inf_r}$ when a is in INF. ❑

3.2.30 Definition (Open INF). A possibly open term a is in *open instantiation normal form* if and only if there does not exist an E_3 and z such that $a = E_3.z$. We write $a\ \mathsf{inf}^\circ_\mathsf{r}$ when a is in open INF. ❑

INF and open INF agree on closed terms, but not necessarily on open ones. For example, if there does not exist E_3, E_3', z, x, u, a such that $a = E_3.\mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ E_3'.z$ and $a \neq E_3'.\mathbf{letrec}_0\ \ z = \lambda x.a\ \ \mathbf{in}\ \ E_3'.z$ then a cannot perform an inst or instrec reduction and is in INF. However, it may still be the case that for some E_3'' and z that $a = E_3''.z$ as long as $z \notin \mathrm{hb}(E_3'')$ and therefore a is not in open INF.
A useful property of instantiation normal forms is that they are preserved by removing a surrounding E_3 context, the proof of which follows easily by proving the contrapositive.

3.2.31 Lemma ($\mathsf{inf}^\circ_\mathsf{r}$ Preserved by E_3 Stripping). *For any evaluation context E_3, if $E_3.a\ \mathsf{inf}^\circ_\mathsf{r}$ then $a\ \mathsf{inf}^\circ_\mathsf{r}$* ❑

To prove that we can reach an instantiation normal form from any $\lambda_{r'}$ term by reduction, we observe that the number of variables above lambdas decreases with every instantiation. Therefore, we define the function $\mathrm{instvar}[e]$ in Definition 3.2.32 and prove that this is monotonically decreasing w.r.t. instantiation reductions to obtain an "INF reachability" result.

3.2.32 Definition ($\mathrm{instvar}[-]$). $\mathrm{instvar}[a]$ denotes the number of potential instantiations that a can do.

$$\begin{array}{rcl}
\mathrm{instvar}[z] & = & 1 \\
\mathrm{instvar}[n] & = & 0 \\
\mathrm{instvar}[()] & = & 0 \\
\mathrm{instvar}[\pi_r\ a] & = & \mathrm{instvar}[a] \\
\mathrm{instvar}[(a\ a')] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a'] \\
\mathrm{instvar}[\lambda^j x.a] & = & 0 \\
\mathrm{instvar}[a\ a'] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a'] \\
\mathrm{instvar}[\mathbf{let}_m\ \ z = a\ \ \mathbf{in}\ \ a'] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a'] \\
\mathrm{instvar}[\mathbf{letrec}_m\ \ z = \lambda x.a\ \ \mathbf{in}\ \ a'] & = & \mathrm{instvar}[a']
\end{array}$$
❑

3.2.33 Lemma ($\mathrm{instvar}[-]$ Properties). *For all $\lambda_{r'}$ terms a and a'*

1. a *r'val* $\Longrightarrow \mathrm{instvar}[a] = 0$
2. $a \xrightarrow{insts}_{r'} a' \Longrightarrow \mathrm{instvar}[a'] = \mathrm{instvar}[a] - 1$

Proof. First prove 1: For $\mathrm{instvar}[u]$ to be non-zero, there must be at least one occurrence of a variable that is not under a lambda binding. By the definition of the forms of values, this cannot be the case.

Now prove 2: Assume $a \xrightarrow{\mathrm{inst}}_{r'} a'$ and prove $\mathrm{instvar}[a'] = \mathrm{instvar}[a] - 1$. One of the following must hold:

$$\exists\ E_3, E_3', z, u.\ a = E_3.\mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ E_3'.z \tag{3.12}$$

$$\exists\ E_3, E_3', z, x, a.\ a = E_3.\mathbf{letrec}_0\ \ z = \lambda x.a\ \ \mathbf{in}\ \ E_3'.z \tag{3.13}$$

Both cases are similar so just consider 3.12. We must prove

$$\mathrm{instvar}[E_3.\mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ E_3'.u] = \mathrm{instvar}[E_3.\mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ E_3'.z] -\ 1$$

which is true if $\mathrm{instvar}[u] = \mathrm{instvar}[z] - 1$, which holds if $\mathrm{instvar}[u] = 0$, which is assured by our first observation.

❑

3.2.34 Lemma (INF Reachability). *For all closed a, if $\mathrm{wf}[a]$ then there exists a' such that* $a \xrightarrow{insts}{}^{*}_{r'} a' \ \wedge\ a'$ $\mathsf{inf_r}$

Proof. Assume a closed and $\mathrm{wf}[a]$. If a does not match the LHS of an inst or instrec rule then we are done, so suppose that it does. By $\mathrm{instvar}[-]$ Properties (Lemma 3.2.33) there can only be finitely many inst or instrec reductions, say n. Thus after n instantiation reductions we arrive at a term a', for which it must hold that a' does not match the LHS of inst or instrec and thus a' $\mathsf{inf_r}$ as required. ❑

We are nearly ready to prove the upper quadrant of the CR-simulation diagram. Before we can though, we need to establish a converse of Lemma 3.2.17, although the converse will not hold directly. The Lemma states that values are preserved by $[\![-]\!]^{\Phi}$, the failure of its converse, that if $[\![\ a\ ]\!]^{\Phi}$ is a value then a is a value, is demonstrated by the following example:

$$[\![\ \mathbf{let}\ x = 3\ \ \mathbf{in}\ \ z\ ]\!]^{\Phi} = 3$$

The result 3 is a value, but **let** $x = 3$ **in** z is not. The extra requirement needed is that a is in INF. The following lemma proves this result.

3.2.35 Lemma ($[\![-]\!]^\Phi$ Source-Value Property). *For all* $\lambda_{r'}$ *expressions* a*, the following holds:*

$$\mathit{wf}[a] \ \wedge \ a \ \ \mathsf{inf}^\circ_\mathsf{r} \ \wedge \ \Phi \blacktriangleleft a \ \wedge \ [\![a]\!]^\Phi \ \mathit{cval} \implies a \ \ \mathit{r'val}$$

Proof. We prove by induction on a. In the identifier case, the term is not in INF. The n and $()$ cases are immediate. $\lambda x.a$ and $\lambda^z x.a$ both result in values when $[\![-]\!]^\Phi$ is applied, but are values themselves. The (a_1, a_2) case follows by induction (and that the subterms are well-formed). The $\pi_r \ a$ and $a_1 \, a_2$ cases are immediate as the action of $[\![-]\!]^\Phi$ on them does not produce a value. The function case is also immediate as $[\![-]\!]^\Phi$ produces a function which is a value. In the **let**$_1$ and **letrec**$_1$ cases, applying $[\![-]\!]^\Phi$ does not produce a value. This leaves the **let**$_0$ and **letrec**$_0$ cases, for which we just show **let**$_0$ as **letrec**$_0$ is similar:

➤*Case* **let**$_0$ $z = a_1$ **in** a_2 **:** Assume the term is well-formed, then the subterms are well-formed and a_1 r'val. Assume the term is in open INF, then a_2 $\mathsf{inf}^\circ_\mathsf{r}$. Assume $\Phi \blacktriangleleft$ **let**$_0$ $z = a_1$ **in** a_2, then by Environment Properties (iii) (Proposition 3.2.15) $\Phi, z \mapsto [\![a_1]\!]^\Phi \blacktriangleleft a_2$. We have to prove:

$$[\![\mathbf{let}_0 \ \ z = a_1 \ \ \mathbf{in} \ \ a_2]\!]^\Phi = [\![a_2]\!]^{\Phi \, , \, z \mapsto [\![a_1]\!]^\Phi}$$

is an r-value, which follows by induction on a_2.

❑

We can now prove the upper quadrant of the cr-simulation.

3.2.36 Lemma (c-r' Correspondence). *If* a *closed and* $\mathit{wf}[a]$ *and* $[\![a]\!]^\varnothing \longrightarrow_c e'$ *then there exists* a', a'' *such that* $a \xrightarrow{insts}{}^*_{r'} a'' \longrightarrow_{r'} a'$ *and* a'' inf_r *and either*

(i) $e' = [\![a']\!]^\varnothing$*; or*

(ii) there exists e'' *such that* $e' \longrightarrow_c e''$ *and* $e'' = [\![a']\!]^\varnothing$*.*

Proof. We generalise to open terms and claim that it is sufficient to prove the following: If $\mathrm{wf}[a]$ and $\Phi \blacktriangleleft a$ and a $\mathsf{inf}^\circ_\mathsf{r}$ and $[\![a]\!]^\Phi \longrightarrow_c e'$ then there exists a' such that $a \longrightarrow_{r'} a'$ and one of the following hold

(i) $e' = [\![a']\!]^\Phi$; or

(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![a']\!]^{\Phi}$.

First let us show that this is sufficient: assume the above proposition and a closed $\wedge$ $\mathrm{wf}[a] \wedge [\![a]\!]^{\varnothing} \longrightarrow_c e'$ then we are required to prove that there exists an a' and a'' such that (3.14) $a \xrightarrow{\text{insts}}{}^{*}_{r'} a''$; (3.15) $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi)$; (3.16) $a'' \longrightarrow_{r'} a'$; (3.17) a'' $\mathsf{inf_r}$ and either (i) or (ii) hold. By INF Reachability (Lemma 3.2.34) there exists an a'' to satisfy 3.14 and 3.17, thus taking $\Phi = \varnothing$ in the generalised claim and applying modus ponens we have that there exists an a' such that $a'' \longrightarrow_{r'} a'$ and (i) or (ii) of the lemma hold. This satisfies the remaining proof obligations.

We prove the generalised claim by induction on the structure of a. In every case except one subcase of application (where recursive functions are considered) we show case (i) holds.

➤*Case* z **:** $\neg(z\ \mathsf{inf_r^\circ})$.

➤*Case* $n; ()$ **:** $[\![n]\!]^{\Phi} = n$ which does not reduce under λ_c. $[\![()]\!]^{\Phi} = ()$ which does not reduce under λ_c.

➤*Case* (a_1, a_2) **:** Assume $\mathrm{wf}[(a_1, a_2)] \wedge (a_1, a_2)\ \mathsf{inf_r^\circ} \wedge [\![(a_1, a_2)]\!]^{\Phi} \longrightarrow_c e'$ and prove that there exists an a' such that $(a_1, a_2) \longrightarrow_{r'} a' \wedge e' = [\![a']\!]^{\Phi}$. We proceed by case split on the reductions of $[\![(a_1, a_2)]\!]^{\Phi}$.

➤*Case* $[\![(a_1, a_2)]\!]^{\Phi} \longrightarrow_c (e_1', [\![a_2]\!]^{\Phi})$ **:** It follows that $[\![a_1]\!]^{\Phi} \longrightarrow_c e_1'$. By $\mathrm{wf}[-]$ definition $\mathrm{wf}[a_1]$. By Environment Properties (Proposition 3.2.15) $\Phi \blacktriangleleft a_1$. By $\mathsf{inf_r^\circ}$ Preserved by E_3 Stripping (Lemma 3.2.31) $a_1\ \mathsf{inf_r^\circ}$. By induction $a_1 \longrightarrow_{r'} a_1' \wedge [\![a_1]\!]^{\Phi} = e_1'$ (*). Thus $(a_1, a_2) \longrightarrow_{r'} (a_1', a_2)$ and we are left to show that the erasure of the RHS of this is equal to $(e_1', [\![a_2]\!]^{\Phi})$: $[\![(a_1', a_2)]\!]^{\Phi} = ([\![a_1']\!]^{\Phi}, [\![a_2]\!]^{\Phi}) = (e_1', [\![a_2]\!]^{\Phi})$ as required.

➤*Case* $[\![(a_1, a_2)]\!]^{\Phi} \longrightarrow_c ([\![a_1]\!]^{\Phi}, e_2')$ **:** Similar to last case, but also uses $[\![-]\!]^{\Phi}$ Source-Value Property (Lemma 3.2.35) to establish a_1 r'val.

➤*Case* $\pi_r\ a$ **:** Assume $\mathrm{wf}[\pi_r\ a] \wedge \pi_r\ a\ \mathsf{inf_r^\circ} \wedge [\![\pi_r\ a]\!]^{\Phi} \longrightarrow_c e'$ and prove that there exists an a' such that $\pi_r\ a \longrightarrow_{r'} a' \wedge e' = [\![a']\!]^{\Phi}$. We proceed by case split on the reductions of $[\![\pi_r\ a]\!]^{\Phi}$.

➤*Case* $[\![\pi_r\ a]\!]^{\Phi} \longrightarrow_c \pi_r\ a'$ **:** Similar to inductive case on pairs.

➤*Case* $[\![\pi_r\ a]\!]^{\Phi} \equiv \pi_r(v_1, v_2) \longrightarrow_c v_r$ **:** It follows that $[\![a]\!]^{\Phi} = (v_1, v_2)$. By $\mathsf{inf_r^\circ}$ Preserved by E_3 Stripping (Lemma 3.2.31) $a\ \mathsf{inf_r^\circ}$. By $[\![-]\!]^{\Phi}$ Source-Value Property

(Lemma 3.2.35) a r'val. By $[\![-]\!]^\Phi$ Outer Value Preservation (Lemma 3.2.21) there exists E_2, u_1, u_2 such that $a = E_2.(u_1, u_2)$. Thus $\pi_r\ a = \pi_r\ E_2.(u_1, u_2) \longrightarrow_{r'} E_2.u_r$. Note that $[\![a]\!]^\Phi = [\![E_2.(u_1, u_2)]\!]^\Phi = ([\![u_1]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}, [\![u_2]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}) = (v_1, v_2)$, thus $[\![E_2.u_r]\!]^\Phi = [\![u_r]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi} = v_r$ as required.

➤*Case* $\lambda^j x.a$ **:** Applying $[\![-]\!]^\Phi$ gives a function which does not reduce.

➤*Case* $a_1\ a_2$ **:** Assume $\mathrm{wf}[a_1\ a_2] \ \wedge\ \Phi \blacktriangleleft a_1\ a_2 \ \wedge\ (a_1\ a_2)\ \mathsf{inf}^\circ_\mathsf{r} \ \wedge\ [\![a_1\ a_2]\!]^\Phi \longrightarrow_c e'$ and prove that there exists an a' such that $a_1\ a_2 \longrightarrow_{r'} a'$ and one of (i) or (ii) hold. We proceed by case split on the reductions of $[\![a_1\ a_2]\!]^\Phi$.

➤*Case* $[\![a_1\ a_2]\!]^\Phi \longrightarrow_c e_1'\ [\![a_2]\!]^\Phi$ **:** Similar to inductive case on pairs ((i) holds).

➤*Case* $[\![a_1\ a_2]\!]^\Phi \longrightarrow_c [\![a_1]\!]^\Phi\ e_2'$ **:** Similar to inductive case on pairs ((i) holds).

➤*Case* $[\![a_1\ a_2]\!]^\Phi \equiv (\lambda x{:}\tau.e)\ v \longrightarrow_c \{v/x\}e$ **:** Thus $[\![a_1]\!]^\Phi = \lambda x{:}\tau.e$ and $[\![a_2]\!]^\Phi = v$. By $\mathsf{inf}^\circ_\mathsf{r}$ Preserved by E_3 Stripping (Lemma 3.2.31) a_1 $\mathsf{inf}^\circ_\mathsf{r}$, so by $[\![-]\!]^\Phi$ Source-Value Property (Lemma 3.2.35) a_1 r'val. As a_1 r'val it follows by $\mathsf{inf}^\circ_\mathsf{r}$ Preserved by E_3 Stripping (Lemma 3.2.31) that a_2 $\mathsf{inf}^\circ_\mathsf{r}$, so by $[\![-]\!]^\Phi$ Source-Value Property (Lemma 3.2.35) a_2 r'val. By Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft \lambda^j x{:}\tau.e$ and $\Phi \blacktriangleleft v$. By $[\![-]\!]^\Phi$ Outer Value Preservation (Lemma 3.2.21) there exists $E_2, j, \tau, \hat{a}$ such that $a_1 = E_2.\lambda^j x{:}\tau.\hat{a}$. There are two cases to consider depending on the form of j.

➤*Case* $j = \cdot$ **:** Thus, $(E_2.\lambda x{:}\tau.\hat{a})\ a_2 \longrightarrow_{r'} E_2.\mathbf{let}\ x = a_2\ \mathbf{in}\ \hat{a}$ and applying $[\![-]\!]^\Phi$ to the RHS gives $[\![\hat{a}]\!]^{\Phi'}$ where $\Phi' = \Phi, \mathcal{E}_c[E_2]^\Phi, x \mapsto [\![a_2]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}$. We are left to show that $[\![\hat{a}]\!]^{\Phi'} = \{v/x\}e$. Do this by expanding $\{v/x\}e$

$$\begin{aligned} \{v/x\}e &= \{[\![a_2]\!]^\Phi/x\}[\![\hat{a}]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi,\, x\mapsto x} \\ &= [\![\hat{a}]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi,\, x\mapsto [\![a_2]\!]^\Phi} && (3.18) \\ &= [\![\hat{a}]\!]^{\Phi'} && (3.19) \end{aligned}$$

3.19 follows from $[\![-]\!]^-$ Environment Properties (i) (Proposition 3.2.20) and 3.18 is true as $\mathrm{fv}(a_2) \notin \mathrm{hb}(E_2)$.

➤*Case* $j = z$ **:** By $[\![-]\!]^-$ Environment Properties (i) (Proposition 3.2.20) $[\![E_2.\lambda^z x.\hat{a}]\!]^\Phi = [\![\lambda^z x.\hat{a}]\!]^{\Phi'}$ where $\Phi' = \Phi, \mathcal{E}_c[E_2]^\Phi$. As $\Phi \blacktriangleleft E_2.\lambda^z x.\hat{a}$ then $\Phi' \blacktriangleleft \lambda^z x.\hat{a}$ by Environment Properties (Proposition 3.2.15) . Thus by the definition of Environment-Term Compatibility (Definition 3.2.7) there exists $\Phi_1, \Phi_2, \hat{e}$ such that

$$\Phi' = \Phi_1, z \mapsto \lambda x.\textbf{letrec } z = \lambda x.\hat{e} \textbf{ in } \hat{e}, \Phi_2 \tag{3.20}$$

$$\hat{e} = [\![\hat{a}]\!]^{\Phi_1, z\mapsto z, x\mapsto x} \tag{3.21}$$

$$\Phi_1, z \mapsto z, x \mapsto x \blacktriangleleft \hat{a} \tag{3.22}$$

We prove point (ii). Observe

$$\begin{array}{rl} & (\lambda x.\textbf{letrec } z = \lambda x.\hat{e} \textbf{ in } \hat{e})v \\ \longrightarrow_c & \textbf{letrec } z = \lambda x.\hat{e} \textbf{ in } \{v/x\}\hat{e} \\ \longrightarrow_c & \{\lambda x.\textbf{letrec } z = \lambda x.\hat{e} \textbf{ in } \hat{e}/z\}\{v/x\}\hat{e} \end{array} \tag{3.23}$$

and

$$(E_2.\lambda^z x.\hat{a})a_2 \longrightarrow_{r'} E_2.\textbf{let } x = a_2 \textbf{ in } \hat{a} \tag{3.24}$$

We are left to show that applying $[\![-]\!]^\Phi$ to 3.24 gives 3.23.

$$\begin{array}{rcl} [\![E_2.\textbf{let } x = a_2 \textbf{ in } \hat{a}]\!]^\Phi & = & [\![\hat{a}]\!]^{\Phi''} \\ \multicolumn{3}{c}{\text{where } \Phi'' = \Phi', x \mapsto [\![a_2]\!]^{\Phi'} \text{ and } \Phi' = \Phi, \mathcal{E}_c[E_2]^\Phi} \end{array}$$

First note that by alpha conversion we can ensure $\text{fv}(a_2) \notin \text{hb}(E_2)$ and therefore $\Phi \blacktriangleleft a_2$. Thus by $[\![-]\!]^-$ Environment Properties (ii) (Proposition 3.2.20) we have $[\![a]\!]^\Phi = [\![a]\!]^{\Phi'}$. By $[\![-]\!]^-$ Environment Properties (i) (Proposition 3.2.20) we have

$$[\![\hat{a}]\!]^{\Phi', x\mapsto [\![a_2]\!]^\Phi} = \{v/x\}[\![\hat{a}]\!]^{\Phi', x\mapsto x}$$

The environment entry $x \mapsto x$ does not have any free variables apart from x, therefore reordering will not invalidate the environment's well-formedness:

$$[\![\hat{a}]\!]^{\Phi_1, z\mapsto\lambda x.\textbf{letrec } z=\lambda x.\hat{e} \textbf{ in } \hat{e}, \Phi_2, x\mapsto x} = [\![\hat{a}]\!]^{\Phi_1, x\mapsto x, z\mapsto\lambda x.\textbf{letrec } z=\lambda x.\hat{e} \textbf{ in } \hat{e}, \Phi_2}$$

We are required to show that this equals 3.23 which we do via the following sequence of deductions whose validity is explained below.

$$\begin{aligned}
& \{v/x\}[\![\,\hat{a}\,]\!]^{\Phi_1\,,\,x\mapsto x\,,\,z\mapsto\lambda x.\mathbf{letrec}\ z=\lambda x.\hat{e}\ \mathbf{in}\ \hat{e}\,,\,\Phi_2} & \\
= {} & \{v/x\}\{\lambda x.\mathbf{letrec}\ z=\lambda x.\hat{e}\ \mathbf{in}\ \hat{e}/z\}[\![\,\hat{a}\,]\!]^{\Phi_1\,,\,x\mapsto x\,,\,z\mapsto z\,,\,\Phi_2} & (3.25)\\
= {} & \{v/x\}\{\lambda x.\mathbf{letrec}\ z=\lambda x.\hat{e}\ \mathbf{in}\ \hat{e}/z\}[\![\,\hat{a}\,]\!]^{\Phi_1\,,\,x\mapsto x\,,\,z\mapsto z} & (3.26)\\
= {} & \{v/x\}\{\lambda x.\mathbf{letrec}\ z=\lambda x.\hat{e}\ \mathbf{in}\ \hat{e}/z\}\hat{e} & (3.27)
\end{aligned}$$

As 3.22 holds so does

$$\Phi_1, x\mapsto x, z\mapsto \lambda x.\mathbf{letrec}\ z=\lambda x.\hat{e}\ \mathbf{in}\ \hat{e} \blacktriangleleft \hat{a}$$

thus by $[\![\,-\,]\!]^-$ Environment Properties (i) (Proposition 3.2.20) 3.25 holds and by $[\![\,-\,]\!]^-$ Environment Properties (ii) (Proposition 3.2.20) 3.26 also holds. Finally, 3.27 holds by 3.21.

➤*Case* $\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2$ **:** Assume wf[$\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2$], $\Phi \blacktriangleleft \mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2$, $(\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2)$ $\mathsf{inf}^\circ_\mathsf{r}$ and $[\![\,\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2\,]\!]^\Phi = [\![\,a_2\,]\!]^{\Phi_1} \longrightarrow_c e'$ where $\Phi_1 = \Phi, z\mapsto [\![\,a_1\,]\!]^\Phi$. By $\mathsf{inf}^\circ_\mathsf{r}$ Preserved by E_3 Stripping (Lemma 3.2.31) a_2 $\mathsf{inf}^\circ_\mathsf{r}$. By definition of wf[$-$] we have wf[$a_2$] $\wedge$ a_1 $\mathsf{r'val}$. By induction $a_2 \longrightarrow_{r'} a_2' \wedge e' = [\![\,a_2'\,]\!]^{\Phi_1}$ (*), thus $\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2 \longrightarrow_{r'} \mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2'$. Now show that applying $[\![\,-\,]\!]^\Phi$ to the RHS of the previous transition gives e': $[\![\,\mathbf{let}_0\ z=a_1\ \mathbf{in}\ a_2'\,]\!]^\Phi = [\![\,a_2'\,]\!]^{\Phi_1} = e'$ follows from (*).

➤*Case* $\mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2$ **:** Assume wf[$\mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2$], $\Phi \blacktriangleleft \mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2$, $(\mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2)$ $\mathsf{inf}^\circ_\mathsf{r}$ and $[\![\,\mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2\,]\!]^\Phi = \mathbf{let}\ z=[\![\,a_1\,]\!]^\Phi\ \mathbf{in}\ [\![\,a_2\,]\!]^\Phi \longrightarrow_c e'$ (*). By wf[$-$] definition wf[a_1] $\wedge$ wf[a_2]. By $\mathsf{inf}^\circ_\mathsf{r}$ Preserved by E_3 Stripping (Lemma 3.2.31) a_1 $\mathsf{inf}^\circ_\mathsf{r}$. By Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft a_1$ and $\Phi, z\mapsto [\![\,a_1\,]\!]^\Phi \blacktriangleleft a_2$.

We case split on the transitions of (*):

➤*Case* $\mathbf{let}\ z=[\![\,a_1\,]\!]^\Phi\ \mathbf{in}\ [\![\,a_2\,]\!]^{\Phi\,,\,x\mapsto x} \longrightarrow_c \mathbf{let}\ z=e_1'\ \mathbf{in}\ [\![\,a_2\,]\!]^{\Phi\,,\,x\mapsto x}$ **:** By induction $a_1 \longrightarrow_{r'} a_1' \wedge e_1' = [\![\,a_1'\,]\!]^\Phi$. Thus $\mathbf{let}_1\ z=a_1\ \mathbf{in}\ a_2 \longrightarrow_{r'} \mathbf{let}_1\ z=a_1'\ \mathbf{in}\ a_2$ and we are left to show that applying $[\![\,-\,]\!]^-$ to the RHS of this results in the RHS of the transition in the case split:

$$\begin{aligned}
[\![\,\mathbf{let}_1\ z=a_1'\ \mathbf{in}\ a_2\,]\!]^\Phi &= \mathbf{let}\ z=[\![\,a_1\,]\!]^\Phi\ \mathbf{in}\ [\![\,a_2\,]\!]^{\Phi\,,\,x\mapsto x}\\
&= \mathbf{let}\ z=e_1'\ \mathbf{in}\ [\![\,a_2\,]\!]^{\Phi\,,\,x\mapsto x}
\end{aligned}$$

as required.

➤*Case* **let** $z = \llbracket a_1 \rrbracket^{\Phi}$ **in** $\llbracket a_2 \rrbracket^{\Phi\,,\,x\mapsto x} \longrightarrow_c \{\llbracket a_1 \rrbracket^{\Phi}/z\}\llbracket a_2 \rrbracket^{\Phi\,,\,x\mapsto x}$ **:** Thus $\llbracket a_1 \rrbracket^{\Phi}$ cval. By $\llbracket - \rrbracket^{\Phi}$ Source-Value Property (Lemma 3.2.35) a_1 r'val, thus **let**$_1$ $z = a_1$ **in** $a_2 \longrightarrow_{r'}$ **let**$_0$ $z = a_1$ **in** a_2. We are left to show that applying $\llbracket - \rrbracket^{-}$ to the RHS of this results in the RHS of the case split:

$$\begin{array}{rcl} \llbracket \textbf{let}_0\ z = a_1\ \textbf{in}\ a_2 \rrbracket^{\Phi} & = & \{\llbracket a_1 \rrbracket^{\Phi}/z\}\llbracket a_2 \rrbracket^{\Phi\,,\,x\mapsto x} \\ & = & \llbracket a_2 \rrbracket^{\Phi\,,\,x\mapsto\llbracket a_1 \rrbracket^{\Phi}} \end{array}$$

where the last step is valid by $\llbracket - \rrbracket^{-}$ Environment Properties (i) (Proposition 3.2.20) .

➤*Case* **letrec**$_0$ $z = \lambda x.a_1$ **in** a_2 **:** Similar to the **let**$_0$ case.

➤*Case* **letrec**$_1$ $z = \lambda x.a_1$ **in** a_2 **:** Similar to the second sub-case of **let**$_1$.

❑

This concludes the proof of the upper quadrant of Figure 3.4. The lower quadrant of the diagram can be stated informally as follows. Suppose $e\ R\ \hat{e}$ via a and a reduces to an instantiation normal form before performing a reduction via rule $\hat{l}$ to a', then $\hat{e}$ can match each instantiation of a and the last reduction $\hat{l}$, provided it is not a zero reduction. In the case where $\hat{l}$ is a zero reduction no further reductions are required for e to equal $\llbracket a' \rrbracket^{\Phi}$. We first show that λ_r can match the instantiation reductions of $\lambda_{r'}$, where the two are related by the $\epsilon[-]$ function.

3.2.37 Lemma (Inst Match Property).

$$wf[a]\ \wedge\ a \xrightarrow{insts}_{r'} a'\ \Longrightarrow\ \exists\ e'.\ \epsilon[a] \xrightarrow{insts}_{r} e'\ \wedge\ e' = \epsilon[a']$$

Proof. We prove by induction on the structure of $a \xrightarrow{\text{insts}}_{r'} a'$ showing only the non-trivial cases:

➤*Case* (inst) **:**

$$\begin{array}{rl} & \epsilon[\textbf{let}_0\ \ z = u\ \ \textbf{in}\ \ E_3.z] \\ = & \textbf{let}\ z = \epsilon[u]\ \textbf{in}\ \epsilon[E_3].\epsilon[z] \\ \longrightarrow_r & \textbf{let}\ z = \epsilon[u]\ \textbf{in}\ \epsilon[E_3].\epsilon[u] \\ = & \epsilon[\textbf{let}\ z = u\ \ \textbf{in}\ \ E_3.u] \end{array}$$

Where the penultimate step is valid by $\epsilon[-]$ Preserves Contexts (Lemma 3.2.26) .

➤*Case* (instrec) : Similar to (inst) case.

➤*Case* (cong) : Assume $\mathrm{wf}[E_3.a]$; $E_3.a \longrightarrow_{r'} E_3.a'$ and $a \xrightarrow{\text{insts}}_{r'} a'$. It follows from Well-Formed Context Decomposition (Lemma 3.2.18) that $\mathrm{wf}[a]$. By induction on a there exists an e' such that $\epsilon[a] \xrightarrow{\text{insts}}_{r'} e'$ and $e' = \epsilon[a']$. As $\epsilon[E_3]$ is a valid λ_r context by $\epsilon[-]$ Preserves Contexts (Lemma 3.2.26) $\epsilon[E_3].\epsilon[a] \longrightarrow_r \epsilon[E_3].e'$. To get the result it is sufficient to prove that $\epsilon[E_3].e' = \epsilon[E_3.a']$. It follows from $\epsilon[-]$ Distributes Over Contexts (Lemma 3.2.25) that $\epsilon[E_3.a'] = \epsilon[E_3].\epsilon[a'] = \epsilon[E_3].e'$ as required. ❑

3.2.38 Lemma (Inst Match Sequence).

$$\mathit{wf}[a] \;\wedge\; a \xrightarrow{\mathit{insts}}{}^{n}_{r'} a' \implies \exists\, e'.\, \epsilon[a] \xrightarrow{\mathit{insts}}{}^{n}_{r} e' \;\wedge\; e' = \epsilon[a']$$

Proof. By induction on the length of the transition sequence (n):

➤*Case* $n = 0$: Immediate.

➤*Case* $n = k$: Assume (3.28) $\mathrm{wf}[a] \;\wedge\; a \xrightarrow{\text{inst}}{}^{k+1}_{r'} a'$ and prove (3.29) $\exists\, e'.\, \epsilon[a] \xrightarrow{\text{insts}}{}^{k+1}_{r} e' \wedge e' = \epsilon[a']$. By 3.28 $\exists\, a''.\ a \xrightarrow{\text{insts}}{}^{k}_{r} a'' \xrightarrow{\text{insts}}_{r} a'$ thus by IH (3.30) $\exists\, e''.\, \epsilon[a] \xrightarrow{\text{insts}}{}^{k}_{r} e'' \wedge e'' = \epsilon[a'']$. Recall that well-formedness is preserved by reduction so $\mathrm{wf}[a'']$. By the above results and Inst Match Property (Lemma 3.2.37) we have (3.31) $\exists\, e'.\, \epsilon[a''] \xrightarrow{\text{insts}}_{r} e' \wedge e' = \epsilon[a']$. Thus by 3.30 and 3.31: $\exists\, e'.\, \epsilon[a] \xrightarrow{\text{insts}}{}^{k+1}_{r'} e' \wedge e' = \epsilon[a']$ as required.

❑

3.2.39 Lemma (r'-r Correspondence).

$$a\ \mathit{closed} \wedge \mathit{wf}[a] \;\wedge\; a \xrightarrow{l}_{r'} a' \;\wedge\; l \neq \mathit{zero} \implies \exists\, e'.\, \epsilon[a] \longrightarrow_r e' \;\wedge\; e' = \epsilon[a']$$

Proof. We generalise to open terms and claim that it is sufficient to prove:

$$\mathrm{wf}[a] \;\wedge\; a \xrightarrow{1}_{r'} a' \;\wedge\; l \neq \mathrm{zero} \implies \exists\, e'.\, \epsilon[a] \longrightarrow_r e' \;\wedge\; e' = \epsilon[a']$$

We prove this by induction on $a \xrightarrow{1}_{r'} a'$.

➤*Case* (proj) : Assume $\mathrm{wf}[\pi_r(E_2.(u_1, u_2))]$. Then $\epsilon[\pi_r(E_2.(u_1, u_2))] = \pi_r\epsilon[E_2].(\epsilon[u_1], \epsilon[u_2])$. By $\epsilon[-]$ Value Preservation (Lemma 3.2.24) $\epsilon[u_1]$ rval and

$\epsilon[u_2]$ rval. Thus by $\epsilon[-]$ Preserves Contexts (Lemma 3.2.26) $\pi_r \epsilon[E_2].(\epsilon[u_1], \epsilon[u_2]) \longrightarrow_r \epsilon[E_2].\epsilon[u_r] = \epsilon[E_2.u_r]$ as required.

➤*Case* (app) : Assume $\mathrm{wf}[(E_2.\lambda x{:}\tau.\hat{a})\ u]$. Then $\epsilon[(E_2.\lambda x{:}\tau.\hat{a})\ u] = (\epsilon[E_2].\lambda x{:}\tau.\epsilon[\hat{a}])\ \epsilon[u]$. By $\epsilon[-]$ Value Preservation (Lemma 3.2.24) $\epsilon[u]$ rval, thus $\epsilon[E_2].((\lambda x{:}\tau.\epsilon[\hat{a}])\ \epsilon[u]) \longrightarrow_r \epsilon[E_2].\mathbf{let}\ x = \epsilon[u]\ \mathbf{in}\ \epsilon[\hat{a}]$. We are left to show that this is equal to the erasure of the RHS of the (app) reduction rule. Performing the erasure of the RHS we get $\epsilon[E_2.\mathbf{let}\ x = u\ \ \mathbf{in}\ \ \hat{a}] = \epsilon[E_2].\mathbf{let}\ x = \epsilon[u]\ \mathbf{in}\ \epsilon[\hat{a}]$, as required.

➤*Case* (inst),(instrec) : Follow directly from Inst Match Property (Lemma 3.2.37) .

➤*Case* (zero),(zerorec) : $l = \mathrm{zero}$.

➤*Case* (cong) : Assume $\mathrm{wf}[E_3.a]$ and $a \longrightarrow_{r'} a'$. By Well-Formed Context Decomposition (Lemma 3.2.18) $\mathrm{wf}[a]$. By induction there exists an e' such that $\epsilon[a] \longrightarrow_r e' \wedge e' = \epsilon[a']$. We are now left to show that the erasure of $E_3.a$ reduces under λ_r to a term that is the erasure of $E_3.a'$. The following reasoning relies on the fact that $\epsilon[E_3]$ is a λ_r context, which can be established by $\epsilon[-]$ Preserves Contexts (Lemma 3.2.26) :

$$\begin{array}{rcl}\epsilon[E_3.a] & = & \epsilon[E_3].\epsilon[a] \\ & \longrightarrow_r & \epsilon[E_3].e' \\ & = & \epsilon[E_3].\epsilon[a'] \\ & = & \epsilon[E_3.a']\end{array}$$

as required.

❑

Putting the c-r' and r'-r correspondence together and using the following lemma (easily proved by inspection), we obtain the cr-simulation result.

3.2.40 Lemma ($\epsilon[-]$ Invariant Under Zeros). *wf*$[a] \wedge a \xrightarrow{\mathrm{zeros}}{}^{*}_{r'} a' \implies \epsilon[a] = \epsilon[a']$ ❑

3.2.41 Lemma (cr Eventually Weak Simulation). *R is an eventually weak simulation from λ_c to λ_r*

Proof. By recalling the definition of eventually weak simulation and expanding the definition of R, assume

$$\exists\ a.\ \mathrm{wf}[a]\ \wedge\ a\ \ \mathsf{closed}\ \wedge\ e_1 = [\![\ a\ ]\!]^{\varnothing}\ \wedge\ e_2 = \epsilon[a] \tag{3.32}$$

$$e_1 \longrightarrow_c e_1' \tag{3.33}$$

We are required to prove that there exists e_2', e_1'' and $n \geq 0$ such that

$$e_2 \longrightarrow_r^* e_2' \tag{3.34}$$

$$e_1' \longrightarrow_c^n e_1'' \tag{3.35}$$

$$\exists\, \hat{a}.\ \mathrm{wf}[\hat{a}] \ \wedge\ \hat{a}\ \ \mathsf{closed}\ \wedge\ e_1'' = [\![\, \hat{a} \,]\!]^{\varnothing}\ \wedge\ e_2' = \epsilon[\hat{a}] \tag{3.36}$$

By $c - r'$ Correspondence (lemma 3.2.36) there exists a' and a'' such that $a \xrightarrow{\text{insts}}{}^{*}_{r'} a'' \ \longrightarrow_{r'}\ a' \ \wedge\ a''\ \ \mathsf{inf_r}$ and either (i) or (ii) of its conclusion holds. Suppose (i) holds, then choose $n = 0$, $e_1'' = e_1'$ and $e_2' = e_2$. Suppose (ii) holds, then there exists the required e_2'' and $n = 1$. In either case choosing $\hat{a} = a'$ confirms $e_1'' = [\![\, a' \,]\!]^{\varnothing}$. It is easily shown from $\lambda_{r'}$ reduction preserves well-formedness (Lemma 3.2.19) that $\mathrm{wf}[\hat{a}]$, and $\hat{a}$ closed as reduction preserves closedness. By Inst Match Sequence (Lemma 3.2.38) there exists an e' such that $\epsilon[a] \xrightarrow{\text{insts}}{}^{*}_{r'} e' \ \wedge\ e' = \epsilon[a'']$.

We now case split on the reduction rule for $a'' \ \longrightarrow_{r'}\ a' = \hat{a}$:

➤*Case* $l = \mathrm{zero}$ **:** By $\epsilon[-]$ Invariant Under Zeros (Lemma 3.2.40) we have $\epsilon[a''] = \epsilon[a']$, thus taking e_2' to be e' satisfies our proof obligation.

➤*Case* otherwise **:** By r'-r Correspondence (Lemma 3.2.39) there exist e_2' such that $\epsilon[a''] \ \longrightarrow_r\ e_2' \ \wedge\ e_2' = \epsilon[a'']$.

❑

An Eventually Weak RC-simulation

We now prove the reverse simulation using a similar process to the one used to prove the CR-simulation. The rôle played by instantiation normal forms is replaced by zero normal forms, with zeros in $\lambda_{r'}$ matching **let**-reductions in λ_c, as shown in Figure 3.5.

We first define Zero Normal Form, establish some properties of it and prove that these forms are always reachable. We do not need to define open and closed ZNFs as we did with INF as the two definitions coincide. That is, a term a can not do an instantiation reduction if and only if the following ZNF condition holds:

3.2.42 Definition (Open ZNF). We say that a possibly open $\lambda_{r'}$ expression is in *open zero normal form* and write $a\ \ \mathsf{znf_r}$ if and only if there does not exist E_3, z, u, a' such that $a = E_3.\mathbf{let}_1\ \ z = u\ \ \mathbf{in}\ \ a'$ ❑

3.2.43 Lemma ($\mathsf{znf_r}$ Preserved by E_3 Stripping). $E_3.a\ \ \mathsf{znf_r} \implies a\ \ \mathsf{znf_r}$

Proof. Proof is easily obtained by proving the contrapositive. ❑

3.2.44 Lemma ($\epsilon[-]$ Source-Value Property). $\mathit{wf}[a] \wedge a\ \ \mathsf{znf_r} \wedge \epsilon[a]\ \mathit{rval} \implies a\ \ \mathit{r'val}$ ❑

3.2.45 Lemma ($\epsilon[-]$ Source Context). *If* $\epsilon[a] = E_3.e$ *and* $a\ \ \mathsf{znf_r}$ *then there exists an* $\hat{E}_3$ *and* $\hat{a}$ *such that* $a = \hat{E}_3.\hat{a}$ *and* $\epsilon[\hat{E}_3] = E_3$.

Proof. Proceed by induction on E_3. We show only a sample of the cases as the rest are similar:

➤*Case* $(v, \_).E_3$ **:** Assume $\epsilon[a] = (v, \_).E_3.e$ and $a\ \ \mathsf{znf_r}$. The only possible form for a is (a_1, a_2) for some a_1 and a_2. Thus $\epsilon[a_1] = v$ and $\epsilon[a_2] = E_3.e$. As a is in open ZNF, a_1 must also be, thus by $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) $a_1\ \ \mathsf{r'val}$. By induction on E_3 there exists $\hat{E}_3$ and $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a} \wedge \epsilon[\hat{E}_3] = E_3$. It follows that $(a_1, a_2) = (a_1, \_).\hat{E}_3.\hat{a} \wedge \epsilon[(a_1, \_).\hat{E}_3] = (v, \_).E_3$.

➤*Case* $\mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ E_3$ **:** Assume $\epsilon[a] = \mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.e$ and $a\ \ \mathsf{znf_r}$. By inspection of the definition of $\epsilon[-]$, a either has the form $\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ or $\mathbf{let}_1\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ for some a_1 and a_2. The latter cannot be the case, as assume that it is, then by $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) $a_1\ \ \mathsf{znf_r}$, but $\epsilon[a_1] = u$ so by $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) $a_1\ \ \mathsf{r'val}$ and so $\mathbf{let}_1\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ is not in open ZNF, a contradiction. We continue considering $a = \mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2$. We have $\epsilon[a_1] = u$, $\epsilon[a_2] = E_3.e$ and as wf$[a]$, $a_1\ \ \mathsf{r'val}$. By induction on E_3 there exists an $\hat{E}_3$ and $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a} \wedge \epsilon[\hat{E}_3] = E_3$. It follows that $a = \mathbf{let}\ z = a_1\ \ \mathbf{in}\ \ \hat{E}_3.\hat{a}$ and $\epsilon[\mathbf{let}\ z = a_1\ \ \mathbf{in}\ \ \hat{E}_3] = \mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3$ as required.

❑

3.2.46 Lemma (ZNF Reachability). *For all closed* a, *if* $\mathit{wf}[a]$ *then there exists* a' *such that* $a \xrightarrow{zero}{}^{*}_{r'} a' \wedge a'\ \ \mathsf{znf_r}$

Proof. To see this we show that all contiguous sequences of (zero)-reductions are finite. Define a metric ones: $\lambda' \to \mathbb{N}$ that counts the number of 1-annotated-**lets** in an expression, then each (zero) reduction strictly reduces this measure. As expressions are finite, our metric is finite-valued and thus reduction sequences consisting only of (zero)-reductions are finite. ❑

For every zero or zerorec reduction that $\lambda_{r'}$ can do, λ_c can match it. We give this result its own lemma as it is used in two places.

3.2.47 Lemma (Zero Match Property).

$$\mathit{wf}[a] \wedge \Phi \blacktriangleleft a \wedge a \xrightarrow{zero}_{r'} a' \implies \exists\, e'.\ [\![\, a \,]\!]^{\Phi} \xrightarrow{let}_{c} e' \wedge e' = [\![\, a' \,]\!]^{\Phi}$$

Proof. We prove by induction on the structure of $a \xrightarrow{\text{zero}}_{r'} a'$:

➤*Case* (zero) : observe

$$\begin{aligned}
\llbracket\, \mathbf{let}_1 \;\; z = u \;\; \mathbf{in} \;\; a \,\rrbracket^{\Phi} \quad &= \quad \mathbf{let}\; z = \llbracket\, u \,\rrbracket^{\Phi} \; \mathbf{in} \; \llbracket\, a \,\rrbracket^{\Phi\,,\, z \mapsto z} \\
&\longrightarrow_c \quad \{\llbracket\, u \,\rrbracket^{\Phi}/z\}\llbracket\, a \,\rrbracket^{\Phi\,,\, z\mapsto z} \\
&= \quad \llbracket\, a \,\rrbracket^{\Phi\,,\, z \mapsto \llbracket\, u \,\rrbracket^{\Phi}} \qquad (3.37) \\
&= \quad \llbracket\, \mathbf{let}_0 \;\; z = u \;\; \mathbf{in} \;\; a \,\rrbracket^{\Phi}
\end{aligned}$$

where step 3.37 is allowed by $\llbracket - \rrbracket^{-}$ Environment Properties (i) (Proposition 3.2.20) .

➤*Case* (zerorec) : Observe:

$$\begin{aligned}
\llbracket\, \mathbf{letrec}_1 \;\; z = \lambda x.a \;\; \mathbf{in} \;\; a' \,\rrbracket^{\Phi} \quad &= \quad \mathbf{letrec}\; z = \lambda x.\llbracket\, a \,\rrbracket^{\Phi\,,\, z\mapsto z\,,\, x\mapsto x} \; \mathbf{in} \; \llbracket\, a' \,\rrbracket^{\Phi\,,\, z\mapsto z} \\
&\longrightarrow_c \quad \{\mu(z, x, \llbracket\, a \,\rrbracket^{\Phi\,,\, z\mapsto z\,,\, x\mapsto x})/z\}\llbracket\, a' \,\rrbracket^{\Phi\,,\, z\mapsto z} \qquad (3.38)
\end{aligned}$$

and

$$\begin{aligned}
\llbracket\, \mathbf{letrec}_0 \;\; z = \lambda x.a \;\; \mathbf{in} \;\; a' \,\rrbracket^{\Phi} \quad &= \quad \llbracket\, a' \,\rrbracket^{\Phi\,,\, z\mapsto \llbracket\, \mu(z,x,a) \,\rrbracket^{\Phi}} \\
&= \quad \{\llbracket\, \mu(z,x,a) \,\rrbracket^{\Phi}/z\}\llbracket\, a' \,\rrbracket^{\Phi\,,\, z\mapsto z} \qquad (3.39)
\end{aligned}$$

It is easily shown that $\llbracket\, \mu(z,x,a) \,\rrbracket^{\Phi} = \mu(z, x, \llbracket\, a \,\rrbracket^{\Phi\,,\, z\mapsto z\,,\, x\mapsto x})$ which makes 3.38 and 3.39 equal, as required.

➤*Case* (cong) : Assume wf$[E_3.a]$; $\Phi \blacktriangleleft E_3.a$; $E_3.a \xrightarrow{\text{zero}}_{r'} E_3.a'$. Notice that $\llbracket\, E_3.a \,\rrbracket^{\Phi} = \llbracket\, E_3 \,\rrbracket^{\Phi}.\llbracket\, a \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}}$. By Environment Properties (iii) (Proposition 3.2.15) $\Phi\,,\mathcal{E}_c[E_3]^{\Phi} \blacktriangleleft a$. By Well-Formed Context Decomposition (Lemma 3.2.18) wf$[a]$. By induction there exists an e'' such that $\llbracket\, a \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}} \longrightarrow_c e'' \wedge e'' = \llbracket\, a' \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}}$. We are required to prove that there exists an e' such that $\llbracket\, E_3.a \,\rrbracket^{\Phi} \xrightarrow{\text{let}}_c e'$ and $e' = \llbracket\, E_3.a' \,\rrbracket^{\Phi}$. By $\llbracket - \rrbracket$ Preserves Contexts (Lemma 3.2.23) there exists E such that $\llbracket\, E_3 \,\rrbracket^{\Phi} = E$. Taking $e' = e''$ by (cong) rule $E.\llbracket\, a \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}} \xrightarrow{\text{let}}_c E.e''$. Finally, $\llbracket\, E_3.a' \,\rrbracket^{\Phi} = \llbracket\, E_3 \,\rrbracket^{\Phi}.\llbracket\, a' \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}}$ by $\llbracket - \rrbracket^{-}$ Distribution Over Contexts (Lemma 3.2.22) and $\llbracket\, E_3 \,\rrbracket^{\Phi}.\llbracket\, a' \,\rrbracket^{\Phi\,,\mathcal{E}_c[E_3]^{\Phi}} = E.e''$ as required.

❑

3.2.48 Lemma (Zero Match Sequence).

$$wf[a] \;\wedge\; \Phi \blacktriangleleft a \;\wedge\; a \xrightarrow{\text{zero}}{}^{n}_{r'} a' \implies \exists\, e'.\; \llbracket\, a \,\rrbracket^{\Phi} \xrightarrow{\text{let}}{}^{n}_{c} e' \;\wedge\; e' = \llbracket\, a' \,\rrbracket^{\Phi}$$

Proof. Proceed by induction on the length of transitions:

➤*Case* $n = 0$ **:** Immediate.

➤*Case* $n = k{+}1$ **:** Assume $\mathrm{wf}[a] \wedge \Phi \blacktriangleleft a \wedge a \xrightarrow{\mathrm{zero}}{}_{r'}^{k+1} a'$. Clearly there exists an a'' such that $a \xrightarrow{\mathrm{zero}}{}_{r'}^{k} a''$. By induction there exists an e' such that $[\![\, a \,]\!]^{\Phi} \xrightarrow{\mathrm{let}}{}_{c}^{k} e' \wedge e' = [\![\, a'' \,]\!]^{\Phi}$ (*). By $- \blacktriangleleft -$ is Closed Under Reduction (Lemma 3.2.8) $\Phi \blacktriangleleft a''$. By Zero Match Property (Lemma 3.2.47) there exists an e'' such that $[\![\, a'' \,]\!]^{\Phi} \xrightarrow{\mathrm{let}}{}_{c} e'' \wedge e'' = [\![\, a' \,]\!]^{\Phi}$ (**). By (*), (**) we have the result.

❑

3.2.49 Lemma (r-r' Correspondence)**.**

a *closed* $\wedge$ $\mathit{wf}[a] \wedge \epsilon[a] \longrightarrow_r e' \implies \exists\, a', a''.\ a \xrightarrow{\mathrm{zero}}{}_{r'}^{*} a'' \longrightarrow_{r'} a' \wedge a''$ $\mathsf{znf_r} \wedge e' = \epsilon[a']$

Proof. We generalise to open terms and claim that it is sufficient to prove:

$$\mathrm{wf}[a] \ \wedge\ a\ \ \mathsf{znf_r}\ \wedge\ \epsilon[a] \longrightarrow_r e' \implies \exists\, a'.\ a \longrightarrow_{r'} a' \ \wedge\ e' = \epsilon[a']$$

Let us show that this is sufficient. Suppose a closed, $\mathrm{wf}[a]$, and $\epsilon[a] \longrightarrow_r e'$ then by ZNF Reachability (Lemma 3.2.46) there exists an a'' such that $a \xrightarrow{\mathrm{zero}}{}_{r'}^{*} a'' \wedge a''$ $\mathsf{znf_r}$. As reduction can only reduce the number of free variables a'' closed and by $\lambda_{r'}$ reduction preserves well-formedness (Lemma 3.2.19) $\mathrm{wf}[a'']$. It thus follows from our generalised claim that there exists an a' such that $a'' \longrightarrow_{r'} a' \wedge e' = \epsilon[a']$. The a' and a'' that we have demonstrated the existence of satisfy the conclusion of our original claim.

We prove the generalised claim by induction on a. The terms $z, ()$ and n are left unchanged by $\epsilon[-]$ and do not reduce under λ_r. $\epsilon[\lambda^j x.a]$ is a function which does not reduce under λ_r. The pair case is just an application of the IH using Well-Formed Context Decomposition (Lemma 3.2.18) and $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) . The rest of the cases follow:

➤*Case* $\pi_r\ a$ **:** Assume $\mathrm{wf}[\pi_r\ a]$, $(\pi_r\ a)$ $\mathsf{znf_r}$ and $\epsilon[\pi_r\ a] \longrightarrow_r e'$. By $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) a r'val. By definition of $\mathrm{wf}[-]$, $\mathrm{wf}[a]$. By $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) a $\mathsf{znf_r}$. Observe $\epsilon[\pi_r\ a] = \pi_r\epsilon[a]$ and case split on the reductions of this:

➤*Case* $\pi_r \epsilon[a] \longrightarrow_r \pi_r\ e'$ **:** Thus $\epsilon[a] \longrightarrow_r e'$. By induction $a \longrightarrow_r a' \wedge e' = \epsilon[a']$, thus $\pi_r\ a \longrightarrow_{r'} \pi_r\ a' \wedge \pi_r\ e' = \epsilon[\pi_r\ a']$ as required.

➤*Case* $\pi_r \epsilon[a] = \pi_r\ E_2.(v_1, v_2) \longrightarrow_r E_2.u_r$ **:** By this case split $\epsilon[a] = E_2.(v_1, v_2)$ (*). By $\epsilon[-]$ Outer Value Preservation (Lemma 3.2.27) there exists $\hat{E_2}, u_1, u_2$ such that $a = \hat{E_2}.(u_1, u_2)$ (**). Thus $\pi_r\ \hat{E_2}.(u_1, u_2) \longrightarrow_{r'} \hat{E_2}.u_r$. We are left to show that $\epsilon[\hat{E_2}.u_r] = E_2.v_r$. By (*) and (**) $\epsilon[\hat{E_2}] = E_2$ and $\epsilon[u_r] = v_r$, thus $\epsilon[\hat{E_2}.u_r] = \epsilon[\hat{E_2}].\epsilon[u_r] = E_2.v_r$ as required.

➤*Case* $a_1\ a_2$ **:** Assuming that (3.40) $\mathrm{wf}[a_1\ a_2]$, (3.41) $a_1\ a_2\ \ \mathsf{znf_r}$ and (3.42) $\epsilon[a_1\ a_2] \longrightarrow_r e'$, we can derive immediately that (3.43) $\mathrm{wf}[a_1] \wedge \mathrm{wf}[a_2]$ and by (3.44) $a_1\ \ \mathsf{znf_r}$.

We case split on the reduction 3.42:

➤*Case* $\epsilon[a_1]\,\epsilon[a_2] \longrightarrow_r e_1'\,\epsilon[a_2]$ **:** Inductive.

➤*Case* $\epsilon[a_1]\,\epsilon[a_2] \longrightarrow_r \epsilon[a_1]\,e_2'$ **:** Inductive.

➤*Case* $\epsilon[a_1]\,\epsilon[a_2] \equiv (E_2.\lambda x{:}\tau.e)\,v \longrightarrow_r E_2.\mathbf{let}\ x = v\ \ \mathbf{in}\ \ e$ **:** By well-formedness definition $\mathrm{wf}[a_1] \wedge \mathrm{wf}[a_2]$. By $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) $a_1\ \ \mathsf{znf_r}$. By $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) $a_1\ \ \mathsf{r'val}$. By $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) $a_2\ \ \mathsf{inf_r^\circ}$. By $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) $a_2\ \ \mathsf{r'val}$. By $\epsilon[-]$ Outer Value Preservation (Lemma 3.2.27) a_1 is of the form $\hat{E_2}.\lambda x{:}\tau.a$. Therefore $\epsilon[a_1] = \epsilon[\hat{E_2}.\lambda x{:}\tau.a] = \lambda x{:}\tau.\epsilon[a]$. By reduction rules $a_1\ a_2 = (\hat{E_2}.\lambda x{:}\tau.a)\ a_2 \longrightarrow_{r'} E_2.\mathbf{let}_0\ \ x = a_2\ \ \mathbf{in}\ \ a$. Then show that erasing this gives the desired result:

$$\epsilon[E_2.\mathbf{let}_0\ \ x = a_2\ \ \mathbf{in}\ \ a] = \epsilon[E_2].\mathbf{let}\ x = \epsilon[a_2]\ \mathbf{in}\ \epsilon[a]$$

We are left to show that $v = \epsilon[a_2]$, which is true by case split assumptions.

➤*Case* $\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ **:** This case proceeds by case analysis on the reductions of $\epsilon[\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2]$. There are two inductive cases, one in which $\epsilon[a_1]$ reduces and the other where $\epsilon[a_2]$ reduces. In both cases we use $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) to establish the open ZNF property of a_1 and a_2 and then proceed by induction. The last possibility is for the term to reduce by doing an instantiation of z. In this case there exists E_3, u such that $(\mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.z) = \epsilon[\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2]$, and we are left to show that there exists an E_3' such that $a_2 = E_3'.z$, which is assured by $\epsilon[-]$ Source Context (Lemma 3.2.45) .

➤*Case* $\mathbf{let}_1\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ **:** This case proceeds by case splitting on the reductions of $\epsilon[\mathbf{let}_1\ \ z = a_1\ \ \mathbf{in}\ \ a_2]$. The first case is when $\epsilon[a_1]$ reduces, which goes by induction on a_1 after using $\mathsf{znf_r}$ Preserved by E_3 Stripping (Lemma 3.2.43) to establish $a_1\ \ \mathsf{znf_r}$. The other possible reduction, occurring when a_1 is a value, say u, is a zero reduction. However, as $a_1\ \ \mathsf{znf_r}$, by $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) $a_1\ \ \mathsf{r'val}$, but then $\mathbf{let}_1\ \ z = u\ \ \mathbf{in}\ \ a_2$ is in open ZNF, a contradiction.

➤*Case* $\mathbf{letrec}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2$, $\mathbf{letrec}_1\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ **:** Similar to their corresponding let cases.

❑

3.2.50 Lemma (r'-c Correspondence). *If a closed and $\mathrm{wf}[a]$ and $a \xrightarrow{l}_{r'} a'$ and $l \neq \mathrm{insts}$ then there exists an e' such that $[\![\, a \,]\!]^{\varnothing} \longrightarrow_c e'$ and either:*

(i) $e' = [\![\, a' \,]\!]^{\varnothing}$

(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![\, a' \,]\!]^{\varnothing}$

Proof. Generalising to open terms it is sufficient to prove: If $\Phi \blacktriangleleft a$ and $\mathrm{wf}[a]$ and $a \xrightarrow{1}_{r'} a'$ and $l \neq \mathrm{insts}$ then there exists e' such that $[\![\, a \,]\!]^{\Phi} \longrightarrow_c e'$ and either:

(i) $e' = [\![\, a' \,]\!]^{\Phi}$

(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![\, a' \,]\!]^{\Phi}$

We prove by induction on $a \xrightarrow{1}_{r'} a'$., showing (i) holds in each case apart from one sub-case of (app).

➤*Case* (proj) **:** Assume $\Phi \blacktriangleleft \pi_r(E_2.(u_1, u_2))$ and $\mathrm{wf}[\pi_r(E_2.(u_1, u_2))]$. Note that

$$[\![\, \pi_r(E_2.(u_1, u_2)) \,]\!]^{\Phi} = \pi_r([\![\, u_1 \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}, [\![\, u_2 \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}) \tag{3.45}$$

and $[\![\, E_2.u_r \,]\!]^{\Phi} = [\![\, u_r \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}$. Our obligation is to show that 3.45 reduces to $[\![\, u_r \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}$.

From our assumptions we know $\Phi \blacktriangleleft E_2.(u_1, u_2)$ thus $\Phi \blacktriangleleft E_2.u_r$. By Environment Properties (i) (Proposition 3.2.15) $\Phi\ ,\mathcal{E}_c[E_2]^{\Phi} \blacktriangleleft u_r$. By $[\![\, - \,]\!]^{-}$ Value Preservation (Lemma 3.2.17) $[\![\, u_r \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}$ cval. It follows that 3.45 reduces to $[\![\, u_r \,]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^{\Phi}}$ under λ_c

➤*Case* (app) **:** Assume $\Phi \blacktriangleleft (E_2.\lambda^j x{:}\tau.a)\, u$, $\mathrm{wf}[(E_2.\lambda^j x{:}\tau.a)\, u]$ and (3.46) $(E_2.\lambda^j x{:}\tau.a)\, u \longrightarrow_{r'} E_2.\mathbf{let}_0\ \ x = u\ \ \mathbf{in}\ \ a$.

We case split on the form of j.

➤*Case* $j = \cdot$ **:** Applying $[\![-]\!]^\Phi$ to the left-hand side of 3.46 and reduce.

$$\begin{array}{rcl} [\![(E_2.\lambda x{:}\tau.a)\, u]\!]^\Phi & = & (\lambda x{:}\tau.[\![a]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^\Phi,\ x\mapsto x})\, [\![u]\!]^\Phi \\ & \longrightarrow_c & \{[\![u]\!]^\Phi/x\}[\![a]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^\Phi,\ x\mapsto x} \\ & = & [\![a]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^\Phi,\ x\mapsto [\![u]\!]^\Phi} \end{array} \qquad (3.47)$$

The last step uses $[\![-]\!]^-$ Environment Properties (i) (Proposition 3.2.20) . Now apply $[\![-]\!]$ to the right-hand side of 3.46:

$$[\![E_2.\mathbf{let}_0\ \ x = u\ \ \mathbf{in}\ \ a]\!]^\Phi \quad = \quad [\![a]\!]^{\Phi\ ,\mathcal{E}_c[E_2]^\Phi,\ x\mapsto [\![u]\!]^{\mathcal{E}_c[E_2]^\Phi,\ \Phi}}$$

To show this equals 3.47 observe that $\Phi \blacktriangleleft u$ by assumptions and Environment Properties (Proposition 3.2.15) and apply $[\![-]\!]^-$ Environment Properties (i) (Proposition 3.2.20) .

➤*Case* $j = z$ **:** First note that by alpha conversion we can ensure that $x \notin \mathrm{dom}(\Phi) \cup \mathrm{hb}(E_2)$ (*). Applying $[\![-]\!]^\Phi$ to the left-hand side of 3.46 we get

$$[\![(E_2.\lambda^z x{:}\tau.a)\, u]\!]^\Phi = (\Phi'(z))\, [\![u]\!]^\Phi$$

where $\Phi' = \Phi, \mathcal{E}_c[E_2]^\Phi$. By Environment Properties (Proposition 3.2.15) we have

$$\Phi' \blacktriangleleft \lambda^z x.a \quad \text{ and } \quad \Phi \blacktriangleleft u \qquad (3.48)$$

Thus by the definition of environment wellformedness there exist Φ_1, Φ_2 such that

$$\Phi' = \Phi_1, z \mapsto \lambda x.\mathbf{letrec}\ z = \lambda x.[\![a]\!]^{\Phi_1'}\ \mathbf{in}\ [\![a]\!]^{\Phi_1'}, \Phi_2 \qquad (3.49)$$

where $\Phi'_1 = \Phi_1, z \mapsto z, x \mapsto x$. Therefore:

$$\begin{array}{rcll}
(\Phi'(z))\llbracket\, u \,\rrbracket^{\Phi} & \longrightarrow_c & \{\llbracket\, u \,\rrbracket^{\Phi}/x\}(\textbf{letrec}\ z = \lambda x.\llbracket\, a \,\rrbracket^{\Phi'_1}\ \textbf{in}\ \llbracket\, a \,\rrbracket^{\Phi'_1}) & \\
& \longrightarrow_c & \{\mu(z, x, \llbracket\, a \,\rrbracket^{\Phi'_1})/z\}\{\llbracket\, u \,\rrbracket^{\Phi}/x\}\llbracket\, a \,\rrbracket^{\Phi'_1} & \\
& = & \{\mu(z, x, \llbracket\, a \,\rrbracket^{\Phi'_1})/z\}\{\llbracket\, u \,\rrbracket^{\Phi}/x\}\llbracket\, a \,\rrbracket^{\Phi'_1\,,\,\Phi_2} & (3.50) \\
& = & \{\mu(z, x, \llbracket\, a \,\rrbracket^{\Phi'_1})/z\}\{\llbracket\, u \,\rrbracket^{\Phi}/x\}\llbracket\, a \,\rrbracket^{\Phi_1\,,\,z\mapsto z\,,\,\Phi_2\,,\,x\mapsto x} & (3.51) \\
& = & \{\mu(z, x, \llbracket\, a \,\rrbracket^{\Phi'_1})/z\}\llbracket\, a \,\rrbracket^{\Phi_1\,,\,z\mapsto z\,,\,\Phi_2\,,\,x\mapsto\llbracket\, u \,\rrbracket^{\Phi}} & (3.52) \\
& = & \llbracket\, a \,\rrbracket^{\Phi_1\,,\,z\mapsto\mu(z,x,\llbracket\, a \,\rrbracket^{\Phi'_1}),\ \Phi_2\,,\,x\mapsto\llbracket\, u \,\rrbracket^{\Phi}} & (3.53)
\end{array}$$

The derivation step 3.50 holds by $\llbracket - \rrbracket^{-}$ Environment Properties (ii) (Proposition 3.2.20) if $\Phi'_1, \Phi_2 \blacktriangleleft a$; let us show this. As $\Phi_1, z \mapsto z, x \mapsto x \blacktriangleleft a$ and Φ_2 is disjoint from Φ_1 and by 3.49 and alpha conversion x and z are not in Φ_2, it holds that $\Phi'_1, \Phi_2 \blacktriangleleft$. By Environment Properties (Proposition 3.2.15) $\Phi'_1, \Phi_2 \blacktriangleleft a$ as required. The derivation step 3.51 holds by expansion of Φ'_1 and Environment Properties (Proposition 3.2.15) if $\Phi_1\ ,\ z \mapsto z\ ,\ \Phi_2\ ,\ x \mapsto x \blacktriangleleft$. To show this it suffices to note that x is not free in the codomain of Φ_2 (as Φ_2 is generated from E_2 in which x is not bound). The derivation step 3.52 holds by $\llbracket - \rrbracket^{-}$ Environment Properties (i) (Proposition 3.2.20) if

$$\Phi_1\ ,\ z \mapsto z\ ,\ \Phi_2\ ,\ x \mapsto x \blacktriangleleft a \tag{3.54}$$

and $\Phi_1\ ,\ z \mapsto z\ ,\ \Phi_2\ ,\ x \mapsto \llbracket\, u \,\rrbracket^{\Phi} \blacktriangleleft$. The former holds by reasoning of the previous derivation step. The latter holds by the following reasoning. As 3.54 holds so does $\Phi_1, z \mapsto z, \Phi_2 \blacktriangleleft a$ and taking this fact together with 3.48 and Environment Properties (i) (Proposition 3.2.15) gives the result.

The derivation step 3.53 holds by $\llbracket - \rrbracket^{-}$ Environment Properties (i) (Proposition 3.2.20) if $\Phi_1\ ,\ z \mapsto \mu(z, x, \llbracket\, a \,\rrbracket^{\Phi'_1}),\ \Phi_2\ ,\ x \mapsto \llbracket\, u \,\rrbracket^{\Phi} \blacktriangleleft$; let us show this. Clearly $\Phi_1 \blacktriangleleft$ and $z \notin \mathrm{dom}(\Phi_1)$. By environment well-formedness definition $\Phi_1, z \mapsto z, z \mapsto z \blacktriangleleft a$. Thus $\Phi_1 \blacktriangleleft \mu(z, x, a)$. By Environment Properties (i) (Proposition 3.2.15) $\Phi_1, z \mapsto \llbracket\, \mu(z, x, a) \,\rrbracket^{\Phi_1} \blacktriangleleft$. By 3.54, 3.48 and Environment Properties (i) (Proposition 3.2.15) $\Phi_1, z \mapsto z, \Phi_2, x \mapsto \llbracket\, u \,\rrbracket^{\Phi} \blacktriangleleft$. The last two facts can be combined to show the result.

Applying $\llbracket - \rrbracket^{\Phi}$ to the right-hand side of 3.46 gives:

$$\llbracket\, E_2.\textbf{let}_0\ x = u\ \textbf{in}\ a \,\rrbracket^{\Phi} \quad = \quad \llbracket\, a \,\rrbracket^{\Phi'\,,\,x\mapsto\llbracket\, u \,\rrbracket^{\Phi'}}$$

but as $\Phi \blacktriangleleft u$ by $[\![-]\!]^-$ Environment Properties (ii) (Proposition 3.2.20)

$$[\![E_2.\mathbf{let}_0 \ x = u \ \mathbf{in} \ a]\!]^\Phi \quad = \quad [\![a]\!]^{\Phi' , x \mapsto [\![u]\!]^\Phi}$$

as required.

➤*Case* (inst), (instrec) : $l = \mathrm{inst}$

➤*Case* (zero), (zerorec) : Follows directly from Zero Match Property (Lemma 3.2.47) .

➤*Case* (cong) : Assume $\Phi \blacktriangleleft E_3.a$, $\mathrm{wf}[E_3.a]$, $E_3.a \longrightarrow_{r'} E_3.a'$ and $a \longrightarrow_{r'} a'$. We can deduce $\Phi \ ,\mathcal{E}_c[E_3]^\Phi \blacktriangleleft a$ by Environment Properties (iii) (Proposition 3.2.15) , and $\mathrm{wf}[a]$ by Well-Formed Context Decomposition (Lemma 3.2.18) . Then by induction there exists e' such that $[\![a]\!]^{\Phi \ ,\mathcal{E}_c[E_3]^\Phi} \longrightarrow_c e'$ and $e' = [\![a']\!]^{\Phi \ ,\mathcal{E}_c[E_3]^\Phi}$. By $[\![-]\!]^-$ Distribution Over Contexts (Lemma 3.2.22) $[\![E_3.a]\!]^\Phi = [\![E_3]\!]^\Phi, [\![a]\!]^{\Phi \ ,\mathcal{E}_c[E_3]^\Phi}$. By $[\![-]\!]$ Preserves Contexts (Lemma 3.2.23) there exists E such that $E = [\![E_3]\!]^\Phi$. By (cong) reduction rule $E.[\![a]\!]^{\Phi \ ,\mathcal{E}_c[E_3]^\Phi} \longrightarrow_c E.[\![a']\!]^{\Phi \ ,\mathcal{E}_c[E_3]^\Phi}$. Using the equalities derived above we can show the result of this reduction to be equal to $[\![E_3.a']\!]^\Phi$ as required.

❑

3.2.51 Lemma (r-c Eventually Weak Simulation). *R is an eventually weak simulation from λ_r to λ_c*

Proof. Recall the definition of eventually weak simulation, expand the definition of R and take $n = 0$. Assume

$$\exists \ a.\ \mathrm{wf}[a] \ \wedge \ a \ \ \mathsf{closed} \ \wedge \ e_1 = [\![a]\!]^\varnothing \ \wedge \ e_2 = \epsilon[a] \tag{3.55}$$

$$e_2 \longrightarrow_r e_2' \tag{3.56}$$

Prove that there exists an e_1' such that

$$e_1 \longrightarrow_c^* e_1' \tag{3.57}$$

$$\exists \ a.\ \mathrm{wf}[a] \ \wedge \ a \ \ \mathsf{closed} \ \wedge \ e_1' = [\![a]\!]^\varnothing \ \wedge \ e_2' = \epsilon[a] \tag{3.58}$$

By r-r' Correspondence (Lemma 3.2.49) there exists a' and a'' such that $a \xrightarrow{\mathrm{zero}}{}^*_{r'} a'' \longrightarrow_{r'} a' \ \wedge \ a'' \ \ \mathsf{znf_r} \ \wedge \ e_1' = \epsilon[a']$. By Zero Match Sequence (Lemma 3.2.48) there exists an e' such that $[\![a]\!]^\varnothing \xrightarrow{\mathrm{let}}{}^*_c e' \ \wedge \ e' = [\![a'']\!]^\varnothing$. Choose a' to be the existentially quantified a in 3.58. We now case split on the reduction rule for $a'' \xrightarrow{1}_{r'} a'$:

➤*Case* $l = \text{insts}$: By $[\![-]\!]^-$ Invariant Under Insts (Lemma 3.2.28) we have $[\![a'']\!]^\varnothing = [\![a']\!]^\varnothing$, thus taking e_1' to be e' satisfies our proof obligation.

➤*Case* otherwise : By r'-c Correspondence (Lemma 3.2.50) there exists e'' such that $[\![a'']\!]^\varnothing \longrightarrow_c e''$ and one of its conclusions (i) or (ii) must hold; consider each in turn.

(i) $e'' = [\![a']\!]^\varnothing$: in this case our proof obligation is immediately satisfied by taking $e_1' = e''$.

(ii) $\exists\ e'''.e'' \longrightarrow_c e_1''' \wedge e''' = [\![a']\!]^\varnothing$: in this case $e_1 \longrightarrow_c^* e_1'''$ so our proof obligations are satisfied by taking $e_1' = e'''$.

❏

3.2.4 Equivalence

Having demonstrated an eventually weak bisimulation between λ_c and λ_r we now wish to use that relation to establish observational equivalence. The bisimulation tells us how terms reduced under λ_r and λ_c are related. However, because the bisimulation is weak it does not tell us anything about how termination behaviour is related between the two calculi. We must show that the termination of expressions coincides for both systems in order to show that the two are observationally equivalent. Figure 3.6 shows diagrammatically how the proof of the main theorem will proceed.

We must first relate $[\![-]\!]_{\mathsf{val}}$ and $[\![-]\!]^-$. The former is used to obtain λ_c values from $\lambda_{r/r'}$ results, while the latter provides the link between λ_c and $\lambda_{r'}$ expressions; we show they are consistent. The main difference is that $[\![-]\!]_{\mathsf{val}}$ uses substitution whereas $[\![-]\!]^-$ uses an environment.

Notation Let σ range over substitutions.

The following definition introduces a function S that builds a substitution from an environment.

$$\begin{tikzcd}
e \arrow[r, "*"{pos=1}, "r"'{pos=1}] & v & \\
a \arrow[u, "\epsilon[-]"] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"'] & a' \arrow[u, "\epsilon[-]"] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"'] \arrow[r, "\text{zero}", "n"{pos=1}, "r'"'{pos=1}] & u \arrow[ul, "\llbracket - \rrbracket^{\varnothing}"'] \arrow[d, "\epsilon[-]"] \\
e \arrow[r, "*"{pos=1}, "c"'{pos=1}] & e'' \arrow[r, "\text{let}", "n"{pos=1}, "c"'{pos=1}] & v'
\end{tikzcd}$$

$$\begin{tikzcd}
e \arrow[r, "*"{pos=1}, "c"'{pos=1}] & v & \\
a \arrow[u, "\llbracket - \rrbracket^{\varnothing}"] \arrow[d, "\epsilon[-]"'] & a' \arrow[u, "\llbracket - \rrbracket^{\varnothing}"] \arrow[d, "\epsilon[-]"'] \arrow[r, "\text{inst}", "n"{pos=1}, "r'"'{pos=1}] & u \arrow[ul, "\epsilon[-]"'] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"] \\
e \arrow[r, "*"{pos=1}, "r"'{pos=1}] & e'' \arrow[r, "\text{inst}", "n"{pos=1}, "c"'{pos=1}] & v'
\end{tikzcd}$$

Figure 3.6: Operational reasoning of r-c equivalence

3.2.52 Definition. Environment-Substitution Correspondence

$$\begin{aligned}
S(\Phi\ ,\ z \mapsto \llbracket\, u \,\rrbracket^{\Phi}) &= S(\Phi)\{\llbracket\, \epsilon[u] \,\rrbracket_{\mathsf{val}}/z\} \\
S(\varnothing) &= \{\}
\end{aligned}$$

❑

As discussed earlier, the simple value collapsing function $\llbracket\, \epsilon[-] \,\rrbracket_{\mathsf{val}}$ and $\llbracket - \rrbracket^{-}$ do not agree on all values, only those of ground type. For values of function type they may not agree as they may differ in their recursive unrollings:

$$\llbracket\, \epsilon[\mathbf{letrec}_0\ \ z =_{\lambda} x.a\ \ \mathbf{in}\ \lambda^{z}x.a] \,\rrbracket_{\mathsf{val}} \quad = \quad \{\lambda x.\mathbf{letrec}\ z = \lambda x.\epsilon[a]\ \mathbf{in}\ \epsilon[a]/z\}(\lambda x.\epsilon[a])$$

but

$$\llbracket\, \mathbf{letrec}_0\ \ z =_{\lambda} x.a\ \ \mathbf{in}\ \lambda^{z}x.a \,\rrbracket^{\varnothing} \quad = \quad \lambda x.\mathbf{letrec}\ z = \llbracket\, a \,\rrbracket^{x\mapsto x\ ,\ z\mapsto z}\ \mathbf{in}\ \llbracket\, a \,\rrbracket^{x\mapsto x\ ,\ z\mapsto z}$$

It turns out that the results of these operators do agree above lambda abstractions, which is sufficient for our purposes as we are interested in contextual equivalence at integer type. This motivates the next definition which is used in following lemma to prove a compatibility result between the two operators.

3.2.53 Definition (Equality on λ Terms up to Functions). We define $=_\lambda$ to be the standard equality relation up to alpha-equivalence, but extended to equate every function. ❏

3.2.54 Lemma (Value Correspondence). *if* $\Phi = \Phi_k$ *where*

$$\begin{array}{lcl} \Phi_0 & = & \varnothing \\ \Phi_{n+1} & = & \Phi_n, x_{n+1} \mapsto [\![\, u_{n+1} \,]\!]^{\Phi_n} \quad \textit{where } fv([\![\, u_{n+1} \,]\!]^{\Phi_n}) = \varnothing \end{array}$$

and $\Phi \blacktriangleleft u$ *and* $\mathit{wf}[u]$ *then* $S(\Phi)[\![\, \epsilon[u] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u \,]\!]^\Phi$

Proof. The proof proceeds by induction on the structure of u.

➤*Case* n; () **:** Immediate.

➤*Case* (u_1, u_2) **:** Assume wf$[(u_1, u_2)]$ and $\Phi \blacktriangleleft (u_1, u_2)$. It is clear from the definition of wf$[-]$ that wf$[u_1] \wedge$ wf$[u_2]$ and by Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft u_1$ and $\Phi \blacktriangleleft u_2$. By induction on u_1 we have $S(\Phi)[\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_1 \,]\!]^\Phi$ and similarly by induction on u_2 we have $S(\Phi)[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_2 \,]\!]^\Phi$. It follows that

$$\begin{array}{rcl} [\![\, (u_1, u_2) \,]\!]^\Phi & =_\lambda & ([\![\, u_1 \,]\!]^\Phi, [\![\, u_2 \,]\!]^\Phi) \\ & =_\lambda & (S(\Phi)[\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}}, S(\Phi)[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}}) \\ & =_\lambda & S(\Phi)[\![\, (\epsilon[u_1], \epsilon[u_2]) \,]\!]_{\mathsf{val}} \end{array}$$

as required.

➤*Case* $\lambda^j x{:}\tau.a$ **:** $\sigma[\![\, \epsilon[\lambda^j x{:}\tau.a] \,]\!]_{\mathsf{val}} = \lambda x{:}\tau.\sigma\epsilon[a]$ and $[\![\, \lambda^j x{:}\tau.a \,]\!]^\Phi$ equals either $\lambda x{:}\tau.[\![\, a \,]\!]^{\Phi\,,\,x \mapsto x}$ or $\Phi(j)$, both are functions (the latter guaranteed to be a function by environment well-formedness) and therefore are equated by $=_\lambda$.

➤*Case* $\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$ **:** Assume wf$[\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2]$ and $\Phi \blacktriangleleft \mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$. We are required to prove

$$S(\Phi)[\![\, \epsilon[\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2] \,]\!]_{\mathsf{val}} =_\lambda [\![\, \mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2 \,]\!]^\Phi$$

By expanding this becomes

$$S(\Phi)\{[\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}}/z\}[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_2 \,]\!]^{\Phi\ ,\ z \mapsto [\![\, u_1 \,]\!]^\Phi}$$

Which holds if and only if

$$S(\Phi, z \mapsto [\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}})[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_2 \,]\!]^{\Phi\ ,\ z \mapsto [\![\, u_1 \,]\!]^\Phi}$$

By Environment Properties (iii) (Proposition 3.2.15) $\Phi \blacktriangleleft u_1$ and $\Phi, z \mapsto [\![\, u_1 \,]\!]^\Phi \blacktriangleleft u_2$ thus by IH applied to u_2

$$S(\Phi, z \mapsto [\![\, u_1 \,]\!]^\Phi)[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_2 \,]\!]^{\Phi\ ,\ z \mapsto [\![\, u_1 \,]\!]^\Phi}$$

It therefore suffices to prove

$$S(\Phi, z \mapsto [\![\, u_1 \,]\!]^\Phi)[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}} =_\lambda S(\Phi, z \mapsto [\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}})[\![\, \epsilon[u_2] \,]\!]_{\mathsf{val}}$$

which holds if $S(\Phi)[\![\, u_1 \,]\!]^\Phi \ =_\lambda\ S(\Phi)[\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}}$. By IH applied to u_1 we have $S(\Phi)[\![\, \epsilon[u_1] \,]\!]_{\mathsf{val}} =_\lambda [\![\, u_1 \,]\!]^\Phi$ and by an easy induction it can be shown that $\mathrm{fv}([\![\, u_1 \,]\!]^\Phi) = \varnothing$, therefore $[\![\, u_1 \,]\!]^\Phi =_\lambda S(\Phi)[\![\, u_1 \,]\!]^\Phi$.

➤*Case* **letrec**$_0$ $z = \lambda x.a$ **in** u_2 **:** This case is similar to the last, but the substitutions for z can be shown in $=_\lambda$ trivially as they are functions.

❑

The following two facts about typing are easily proved by induction on the typing derivation.

3.2.55 Lemma (Typing is Substitutive). $\Gamma \vdash v{:}\tau \ \wedge\ \Gamma, z{:}\tau \vdash e{:}\tau' \implies \Gamma \vdash \{v/z\}e{:}\tau'$ ❑

3.2.56 Lemma ($[\![\, - \,]\!]_{\mathsf{val}}$ Type Preservation). $\Gamma \vdash u{:}\tau \implies \Gamma \vdash [\![\, u \,]\!]_{\mathsf{val}}{:}\tau$ ❑

Proof of The Main Theorem

Proof of Theorem 3.2.1. We begin by proving point 1 of the theorem. First prove:

$$\begin{aligned} &e\ \mathsf{closed} \wedge e \longrightarrow_c^* v_1 \implies \\ &\qquad \exists\, v_2, u.\ e \longrightarrow_r^* v_2 \ \wedge\ \mathrm{wf}[u] \ \wedge\ u\ \mathsf{closed} \ \wedge\ v_1 = [\![\, u \,]\!]^\varnothing \ \wedge\ v_2 = \epsilon[u](*) \end{aligned}$$

Assume e closed and $e \longrightarrow_c^* v_1$, and recall $e\ R\ e$ by R Contains Identity (Lemma 3.2.13) . By c-r Eventually Weak Simulation (Lemma 3.2.41) R is a c-r simulation, thus there exists an e' such that $e \longrightarrow_r^* e'$ and $v_1 \; R \; e'$. Expanding the definition of R in the latter, we are assured that

$$\exists\, a.\ \mathrm{wf}[a] \ \wedge\ a \ \ \mathsf{closed} \ \wedge\ v_1 = [\![\, a \,]\!]^{\varnothing} \ \wedge\ e' = \epsilon[a]$$

We are left to show $e' \longrightarrow_r^* e''$ and e'' rval. By $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) it suffices to prove that there exists an a' such that a' $\mathsf{r'val}$ $\wedge$ $\mathrm{wf}[a']$ $\wedge$ a' $\mathsf{znf_r}$ $\wedge$ $e'' = \epsilon[a']$.

Suppose that a $\mathsf{inf_r}$, then by $[\![-]\!]^{\Phi}$ Source-Value Property (Lemma 3.2.35) a $\mathsf{r'val}$. By $\epsilon[-]$ Value Preservation (Lemma 3.2.24) $\epsilon[a]$ rval as required.

Now suppose that $\neg(a \ \mathsf{inf_r})$ then by INF Reachability (Lemma 3.2.34) there exists an a'' such that $a \longrightarrow_{r'}^* a' \wedge a'$ $\mathsf{inf_r}$. By $\lambda_{r'}$ reduction preserves well-formedness (Lemma 3.2.19) $\mathrm{wf}[a']$ and by $[\![-]\!]^{-}$ Invariant Under Insts (Lemma 3.2.28) $v_1 = [\![\, a' \,]\!]^{\varnothing}$. Thus by $[\![-]\!]^{\Phi}$ Source-Value Property (Lemma 3.2.35) a' $\mathsf{r'val}$. By Inst Match Sequence (Lemma 3.2.38) there exists an e'' such that $e' \longrightarrow_r^* e'' \wedge e'' = \epsilon[a']$ as required.

Now prove the main theorem:

$$\vdash e{:}\mathsf{int} \ \wedge\ e \longrightarrow_c^* n \implies \exists\, v.\ e \longrightarrow_r^* v \ \wedge\ n = [\![\, v \,]\!]_{\mathsf{val}}$$

Assuming $\vdash e{:}\tau \wedge e \longrightarrow_c^* n$ we can derive e closed, thus by (*) we know that there exists a u and v_2 such that $e \longrightarrow_r^* v_2 \wedge \mathrm{wf}[u] \wedge u$ closed $\wedge$ $n = [\![\, u \,]\!]^{\varnothing} \wedge v_2 = \epsilon[u]$.

We are left to show that $n = [\![\, v_2 \,]\!]_{\mathsf{val}}$. By Value Correspondence (Lemma 3.2.54) $[\![\, \epsilon[u] \,]\!]_{\mathsf{val}} = [\![\, u \,]\!]^{\varnothing}$. We are left to show that this value is an integer, for which it suffices to show that one of the values in the equality above types to int, as the only values of type int in λ_c are integers. By type preservation for λ_r $\vdash v_2{:}\mathsf{int}$, thus $\vdash \epsilon[u]{:}\mathsf{int}$ by dint of equality with v_2. By $[\![-]\!]_{\mathsf{val}}$ Type Preservation (Lemma 3.2.56) $\vdash [\![\, \epsilon[u] \,]\!]_{\mathsf{val}}{:}\mathsf{int}$, as required.

Now prove point 2. First prove:

$$e \ \ \mathsf{closed} \wedge e \longrightarrow_r^* v_1 \implies \exists\, v_2, u.\ e \longrightarrow_c^* v_2 \wedge \mathrm{wf}[u] \wedge u \ \ \mathsf{closed} \wedge v_2 = [\![\, u \,]\!]^{\varnothing} \wedge v_1 = \epsilon[u]$$

Assume e closed and $e \longrightarrow_r^* v_1$, and recall $e \; R \; e$ by R Contains Identity (Lemma 3.2.13) . By r-c Eventually Weak Simulation (Lemma 3.2.51) R is a r-c simulation, thus there exists an e' such that $e \longrightarrow_c^* e'$ and $e' \; R \; v_1$. Expanding the definition of R in the latter, we are assured that

$$\exists\, a.\ \mathrm{wf}[a] \ \wedge\ a \ \ \mathsf{closed} \ \wedge\ e' = [\![\, a \,]\!]^{\varnothing} \ \wedge\ v_1 = \epsilon[a]$$

We are left to show $e' \longrightarrow_c^* e''$ and e'' cval. By $[\![-]\!]^\Phi$ Source-Value Property (Lemma 3.2.35) it suffices to prove that there exists an a' such that a' r'val $\wedge$ wf$[a']$ $\wedge$ a' inf$_\mathsf{r}$ $\wedge$ $e'' = [\![a']\!]^\varnothing$.

Suppose that a znf$_\mathsf{r}$ then by $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) a r'val. By $[\![-]\!]^-$ Value Preservation (Lemma 3.2.17) $[\![a']\!]^\varnothing$ cval as required.

Now suppose that $\neg(a$ znf$_\mathsf{r})$ then by ZNF Reachability (Lemma 3.2.46) there exists an a'' such that $a \xrightarrow{\text{zeros}}{}^*_{r'} a' \wedge a'$ znf$_\mathsf{r}$. By $\lambda_{r'}$ reduction preserves well-formedness (Lemma 3.2.19) wf$[a']$ and by $\epsilon[-]$ Invariant Under Zeros (Lemma 3.2.40) $v_1 = \epsilon[a']$. Thus by $\epsilon[-]$ Source-Value Property (Lemma 3.2.44) a' r'val. By Zero Match Sequence (Lemma 3.2.48) there exists an e'' such that $e' \longrightarrow_c^* e'' \wedge e'' = \epsilon[a']$ as required.

Now prove the main theorem:

$$\vdash e{:}\mathsf{int} \wedge e \longrightarrow_r^* v \implies \exists\, n.\ e \longrightarrow_c^* n \wedge n = [\![v]\!]_\mathsf{val}$$

Assume $\vdash e{:}\mathsf{int}$ and $e \longrightarrow_r^* v$ then by the above lemma there exists a v_2 and a u such that $e \longrightarrow_c^* v_2$; wf$[u]$; u closed; $v_2 = [\![u]\!]^\varnothing$; $v = \epsilon[u]$ and u r'val.

We are left to show that $[\![u]\!]_\mathsf{val} = n$. By Value Correspondence (Lemma 3.2.54) $[\![\epsilon[u]]\!]_\mathsf{val} = [\![u]\!]^\varnothing$. We are left to show that this value is an integer, for which it suffices to show that one of the values in the equality above types to int, as the only values of type int in λ_c are integers. By type preservation for λ_r $\vdash v{:}\mathsf{int}$, thus $\vdash \epsilon[u]{:}\mathsf{int}$ by dint of equality with v. By $[\![-]\!]_\mathsf{val}$ Type Preservation (Lemma 3.2.56) $\vdash [\![\epsilon[u]]\!]_\mathsf{val}{:}\mathsf{int}$, as required. ❑

3.3 Observational Equivalence Between λ_d and λ_c

Having shown in the last section that a certain form of contextual equivalence coincides for λ_r and λ_c we now turn to showing a similar result for λ_d and λ_c. This proof closely follows the structure and techniques used in the last section. We therefore concentrate on those parts where the proof differs, simply stating those lemmas that follow in a straightforward way from the corresponding earlier result.

We prove the following theorem

3.3.1 Theorem (Observational Equivalence of λ_d and λ_c). *For all $e \in \lambda$ the following hold:*

1. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_c^* n \implies \exists\, v.\ e \longrightarrow_d^* v \wedge n = [\![v]\!]_\mathsf{val})$
2. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_d^* v \implies \exists\, n.\ e \longrightarrow_c^* n \wedge n = [\![v]\!]_\mathsf{val})$

We borrow verbatim the annotated syntax λ'; the functions $\iota[-]$, $\epsilon[-]$ and $\mathcal{E}_c[-]^-$; and the predicate $\mathrm{wf}[-]$ from the λ_r proof.

As we are ultimately interested only in closed terms, we are free to alter the behaviour of λ_c on open terms so long as it remains the same when restricted to closed terms. We do this by adding identifiers to the set of values for λ_c:

$$v ::= z \mid n \mid () \mid \lambda x{:}\tau.e$$

3.3.2 Definition ($\lambda_{d'}$ Reduction Semantics)**.** This is as defined for $\lambda_{r'}$ except we add *destruct contexts*:

$$R ::= \pi_r \_ \mid \_\, u$$

The (inst) and (instrec) reduction rules are replaced with corresponding ones for destruct time instantiation:

(inst-1)	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.z$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.u$
(inst-2)	$R.E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_2'.z$	$\longrightarrow$	$R.E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_2'.u$
(instrec-1)	$\mathbf{letrec}_0\ z = \lambda x.a\ \mathbf{in}\ E_3.R.E_2.z$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.\lambda^z x.a$
(instrec-2)	$R.E_2.\mathbf{letrec}_0\ z = \lambda x.a\ \mathbf{in}\ E_2'.z$	$\longrightarrow$	$R.E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_2'.\lambda^z x.a$

❑

The definition for environment must be changed to admit the possibility of identifiers as values, but the definition of environment-term compatibility remains unaltered.

3.3.3 Definition (Environment)**.** An *environment* Φ is a list containing pairs whose first component is an identifier and whose second component is a c-value. An environment is well-formed if the following hold:

(i) whenever $(z, e) \in \Phi$ forall $z \in \mathrm{fv}(e).z{\leq_\Phi}x$ where $\leq_\Phi$ is the ordering of the identifiers in Φ.

(ii) all of the first components of the pairs in the list are distinct.

When Φ is well-formed we write Φ ◀. We write $\Phi, z \mapsto v$ for the disjoint extension of Φ forming a new environment. We write $\Phi[z \mapsto v]$ for the environment acting as Φ, but mapping z to v ❑

A key difference with λ_d is that identifiers are values. As a result whenever we instantiate a variable we may obtain another variable. During reduction this chain of variables is followed until a non-variable value is met. To reflect this difference in $\llbracket - \rrbracket^-$ we change variable lookup to follow the chain.

3.3.4 Definition ($\llbracket - \rrbracket^-$)**.** We use the definition from the λ_r case with the following change:

$$\llbracket z \rrbracket^\Phi = \Phi^*(z)$$

where we define Φ^* as the least fixpoint of the monotone operator F:

$$F(\Phi) = \Phi[x \mapsto z \mid \exists\, y.\ \Phi(x) = y\ \wedge\ \Phi(y) = z]$$

❑

Instantiation normal forms change to reflect the destruct time nature of instantiations.

3.3.5 Definition (Instantiation Normal Form (INF))**.** A term a is in *instantiation normal form* (INF) if and only if there does not exist an a' such that $a \xrightarrow{\text{inst}}_{d'} a'$, where inst is inst-1 or inst-2. We write $a\ \ \mathsf{inf_d}$ when a is in INF. ❑

3.3.6 Definition (Open INF)**.** A possibly open term a is in *open instantiation normal form* if and only if there does not exist an E_3, R, E_2 and z such that $a = E_3.R.E_2.z$. We write $a\ \ \mathsf{inf_d^\circ}$ when a is in open INF. ❑

3.3.1 Transforming proofs from λ_r to λ_d

To avoid duplicating tedious proofs we would like to reuse as much reasoning from the λ_r proof as possible. To do this let us first enumerate the entities that have changed from the λ_r proof:

1. identifiers added to values;
2. introduction of destruct contexts R;
3. new instantiation rules; and
4. the definitions of Φ and $\llbracket - \rrbracket^-$ changed.

Most changes to the proofs occur where values are considered, as we must now deal with identifiers, and where reduction is considered, as the instantiation rules have changed, though this second change is not very deep as the rules are still of a similar form.

Lemma No.	Name of Lemma	λ_r proof lemma No.
3.3.7	Well-Formed Contex Decomposition	3.2.18
3.3.8	$\llbracket - \rrbracket$ Distribution Over Contexts	3.2.22
3.3.9	$\llbracket - \rrbracket$ Preserves Contexts	3.2.23
3.3.10	R Contains Identity	3.2.13
3.3.11	$\epsilon[-]$ Value Preservation	3.2.24
3.3.12	$\epsilon[-]$ Distributes Over Contexts	3.2.25
3.3.13	$\epsilon[-]$ Preserves Contexts	3.2.26
3.3.14	$\epsilon[-]$ Source-Value Property	3.2.44
3.3.15	$\epsilon[-]$ Outer Value Preservation	3.2.27
3.3.16	$\mathsf{inf}_\mathsf{r}^\circ$ Preserved by E_3 Stripping	3.2.31
3.3.17	Inst Match Property	3.2.37
3.3.18	Inst Match Sequence	3.2.38
3.3.19	$\epsilon[-]$ Invariant Under Zeros	3.2.40
3.3.20	znf_r Preserved by E_3 Stripping	3.2.43
3.3.21	$\epsilon[-]$ Source Context	3.2.45
3.3.22	ZNF Reachability	3.2.46
3.3.23	Zero Match Property	3.2.47
3.3.24	Zero Match Sequence	3.2.48
3.3.25	$\llbracket - \rrbracket_\mathsf{val}$ Type Preservation	3.2.56

Table 3.1: Lemmas that follow in a straightforward way from the proof given of the corresponding fact in the λ_r-λ_d equivalence proof.

Although destruct contexts have been introduced, notice that every R context is an A_1 context and so little of the reasoning changes. In particular, the $E_3.R.E_2$ context in the new instantiation rules is a particular form of E_3 context.

We might expect some work to be needed due to the change in the definition of environment. However, the conditions for adding elements to it have been weakened (we now allow identifiers to map to other identifiers) and when the new definition of $\llbracket - \rrbracket^\Phi$ looks up identifiers it does so in the "transitive closure" of Φ: $\llbracket z \rrbracket^\Phi = \Phi^*(z)$. This transitive closure of Φ is an environment like that used for the λ_r proof, in the sense that identifiers map to themselves or to a non-identifier value.

Most proofs have a trivial translation from their counter part in the previous section and we merely note these in Table 3.1 without repeating the proof.

3.3.2 Changed Proofs

3.3.26 Lemma (Reduction Preserves Well-formedness). $wf[a] \wedge a \longrightarrow_{d'} a' \implies wf[a']$

Proof. The proof is by induction on $a \longrightarrow_{d'} a'$. All the common cases follow analogously from the λ_r proof, leaving the two inst cases, which are similar. We consider (inst-1): assume $\mathrm{wf}[\mathbf{let}_0 \ \ z = u \ \ \mathbf{in} \ \ E_3.R.E_2.z]$ then $\mathrm{wf}[E_3.R.E_2.z]$ and by definition of well-founded $\mathrm{wf}[u]$, thus by Lemma 3.3.7 $\mathrm{wf}[E_3.R.E_2.u]$. It follows that $\mathrm{wf}[\mathbf{let}_0 \ \ z = u \ \ \mathbf{in} \ \ E_3.R.E_2.u]$ as required. ❑

The properties of $[\![-]\!]^-$ remain mainly the same, however because identifiers are now values we need to specialise the first point of the $[\![-]\!]^-$ environment properties, because the more general statement made in the λ_r proof no longer holds. However, this is sufficient for our use.

3.3.27 Lemma ($[\![-]\!]^-$ Environment Properties).

(i) *If* $\mathrm{wf}[a]$ *and* $\Phi, x \mapsto x \blacktriangleleft a$ *and* $\Phi \blacktriangleleft u$ *and* $\Phi, x \mapsto [\![u]\!]^\Phi, \Phi' \blacktriangleleft$ *then* $\{[\![u]\!]^\Phi/x\}[\![a]\!]^{\Phi \ , \ x \mapsto x \ , \ \Phi'} = [\![a]\!]^{\Phi \ , \ x \mapsto [\![u]\!]^\Phi , \ \Phi'}$

(ii) *If* $\Phi \blacktriangleleft a$ *and* $\Phi, \Phi' \blacktriangleleft a$ *then* $[\![a]\!]^\Phi = [\![a]\!]^{\Phi \ , \ \Phi'}$

(iii) *If* $\Phi_1, \Phi_2, \Phi_3, \Phi_4 \blacktriangleleft a$ *and* $\Phi_1, \Phi_3, \Phi_2, \Phi_4 \blacktriangleleft a$ *then* $[\![a]\!]^{\Phi_1 \ , \ \Phi_2 \ , \ \Phi_3 \ , \ \Phi_4} = [\![a]\!]^{\Phi_1 \ , \ \Phi_3 \ , \ \Phi_2 \ , \ \Phi_4}$.

Proof. Prove (i) by induction on a. The interesting case is the identifier case:

➤*Case* z **:** Assume $\mathrm{wf}[z]$, $\Phi \blacktriangleleft u$, $\Phi, x \mapsto [\![u]\!]^\Phi, \Phi' \blacktriangleleft$ and

$$\Phi, x \mapsto x \blacktriangleleft z \tag{3.59}$$

It suffices to prove $\{[\![u]\!]^\Phi/x\}[\Phi \ , \ x \mapsto x \ , \ \Phi']^*(z) = [\Phi \ , \ x \mapsto [\![u]\!]^\Phi, \ \Phi']^*(z)$. There are two cases to consider

➤*Case* $z = x$ **:** In this case $[\Phi \ , \ x \mapsto x \ , \ \Phi']^*(z) = x$ and thus

$$\{[\![u]\!]^\Phi/x\}[\Phi \ , \ x \mapsto x \ , \ \Phi']^*(z) = [\![u]\!]^\Phi$$

Now suppose $[\![u]\!]^\Phi$ is not an identifier then $[\Phi \ , \ x \mapsto [\![u]\!]^\Phi, \ \Phi']^*(z) = [\![u]\!]^\Phi$ and we are done. Now consider the other possibility, that $[\![u]\!]^\Phi$ is an identifier, say y, then it must be the case that $\Phi^*(y) = y$. We must show that $[\Phi \ , \ x \mapsto y \ , \ \Phi']^*(x) = y$, but this holds if and only if $\Phi^*(y) = y$ which has already been shown.

➤*Case* $z \neq x$ **:** In this case $z \in \mathrm{dom}(\Phi)$ as 3.59 holds. It follows that $[\Phi\ ,\ x \mapsto x\ ,\ \Phi']^*(z) = \Phi^*(z) = [\Phi\ ,\ x \mapsto [\![\, u \,]\!]^{\Phi},\ \Phi']^*(z)$ as required.

❑

3.3.28 Lemma ($[\![\, - \,]\!]^{-}$ Value Preservation)**.** *If* $wf[u]$ *and* $\Phi \blacktriangleleft u$ *then* $[\![\, u \,]\!]^{\Phi}$ *cval*

Proof. The proof is similar to the corresponding λ_r one. The new case is identifiers, which is easily discharged as $[\![\, z \,]\!]^{\Phi} = \Phi^*(z)$ which is a c-value. ❑

We now prove that every instantiation sequence is finite by finding a metric that decreases with each instantiation. For λ_r this was straight forward by observing that the number of identifiers above λ abstractions decreases with each instantiation. This is not true in λ_d as the instantiated value can contain variables above λ abstractions because variables are values. Our approach here is based on the following observations:

1. The form of a term that can undergo an instantiation is $E_3.R.E_2.z$ where z is bound in $E_3.R.E_2$.

2. If in such a term z instantiates to $E_2'.(u_1, u_2)$ and $R = \pi_r\ \_$ then by unique decomposition the next reduction is a projection. On the other hand if z instantiates to $E_2'.\lambda^z.a$ and $R = \_\ u$ then by unique decomposition the next reduction is an application. In either case the term is in INF. If the R does not match the instantiate value then we are stuck and again in INF.

3. If in such a term z instantiates to $E_2'.n$, or it in instantiates to $E_2'.(u_1, u_2)$ when R is an application destruct context or $E_2'.\lambda^z.a$ when R is a projection destruct context then it is stuck.

4. The final option is for z to instantiate to $E_2'.z'$ in which case another instantiation is possible.

It is this last point – a reduction of the form $E_3.R.E_2.z \longrightarrow E_3.R.E_2.z'$ – that requires some consideration. We must show that in any given chain of instantiations there can only be finitely many of this form. To do this we label each let in the program with a unique number in such a way that in any contiguous sequence of instantiations the next let to be instantiated is less than the one before. The following definition labels terms.

3.3.29 Definition ($\mathrm{label}_-(-)$)**.** label the lets in a term with natural numbers:

$$\begin{array}{lcl}
\mathrm{label}_l(x) & = & x \\
\mathrm{label}_l(n) & = & n \\
\mathrm{label}_l(\lambda x.e) & = & \lambda x.\mathrm{label}_l(e) \\
\mathrm{label}_l((e_1, e_2)) & = & \mathrm{label}_l(e_1)\mathrm{label}_{l+|\mathrm{label}_l(e_1)|}(e_2) \\
\mathrm{label}_l(\pi_r\ e) & = & \pi_r \mathrm{label}_l(e) \\
\mathrm{label}_l(e_1\ e_2) & = & \mathrm{label}_l(e_1)\mathrm{label}_{l+|\mathrm{label}_l(e_1)|}(e_2) \\
\mathrm{label}_l(\mathbf{let}_m\ \ z = e_1\ \ \mathbf{in}\ \ e_2) & = & \\
\multicolumn{3}{r}{\mathbf{let}_m^{l+|\mathrm{label}_l(e_1)|} z = \mathrm{label}_l(e_1)\ \mathbf{in}\ \mathrm{label}_{l+|\mathrm{label}_l(e_1)|+1}(e_2)} \\
\mathrm{label}_l(\mathbf{letrec}_m\ \ z = \lambda x.\mathrm{label}_l(e_1)\ \mathbf{in}\ \ e_2) & = & \\
\multicolumn{3}{r}{\mathbf{letrec}_m^{l+|\mathrm{label}_l(e_1)|} z = \lambda x.\mathrm{label}_l(e_1)\ \mathbf{in}\ \mathrm{label}_{l+|\mathrm{label}_l(e_1)|+1}(e_2)}
\end{array}$$

The auxilliary function $|e|$ determines the highest label in e:

$$\begin{array}{lcl}
|z| & = & 0 \\
|n| & = & 0 \\
|\lambda x.e| & = & |e| \\
|(e_1, e_2)| & = & \max(|e_1|)|e_2| \\
|\pi_r\ e| & = & |e| \\
|e_1\ e_2| & = & \max(|e_1|, |e_2|) \\
|\mathbf{let}_m^l z = e_1\ \ \mathbf{in}\ \ e_2| & = & \max(l\ , |e_1|, |e_2|) \\
|\mathbf{letrec}_m^l z = \lambda x.e_1\ \ \mathbf{in}\ \ e_2| & = & \max(l\ , |e_1|, |e_2|)
\end{array}$$

We also define $\mathrm{letlabels}(e)$ to return the set of label identifier pairs, $\{(l_0, z_0), ..., (l_n, z_n)\}$, bound by lets in e.

We extend these functions to contexts with $|\_| = 0$, $\mathrm{label}_l(\_) = \_$ and $\mathrm{letlabels}(\_) = \varnothing$. We write $\mathrm{lhb}(E_3)$ for the set of label identifier pairs that bind around the hole in E_3. ❑

We also need the concept of minimum label:

3.3.30 Definition ($\min(-)$)**.** Define $\min(L)$ to return the label of least order in the identifier label pairs L. When L is the empty set define $\min(\varnothing) = \infty$. ❑

We first prove two auxiliary results.

3.3.31 Lemma. *If* $E_3.R.E_2.z_0 = \mathrm{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0) \wedge E_3.R.E_3.z_0$ *closed then for all* $n \in \mathbb{N}$ *if* $E_3.R.E_2.z_0 \xrightarrow{inst}_{d'} ... \xrightarrow{inst}_{d'} E_3.R.E_2.E_2^1...E_2^n$ *and* $\mathrm{fv}(E_2^n.z_n) = \{x_1, ..., x_m\}$ *then*

$$\{(x_1, l_1'), ..., (x_m, l_m')\} \subseteq lhb(E_3.R.E_2.E_2^1...E_2^{n-1})$$

and

$$l_1, ..., l_m < \min(\mathrm{letlabels}(E_2^n))$$

Proof. We proceed by assuming $E_3.R.E_2.z_0 = \mathrm{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0)$ and $E_3.R.E_2.z_0$ closed and proving the consequence by induction on n, the number of transitions.

➤*Case* $n = 0$ **:** As $E_3.R.E_2.z_0$ closed we have $(z_0, l_0) \in \mathrm{lhb}(E_3.R.E_2)$. As $\min(\mathrm{letlabels}(z_0)) = \min(\varnothing) = \infty$ it follows that $l_0 < \min(\varnothing)$ as required.

➤*Case* $n = k + 1$ **:** Assume $E_3.R.E_2.z_0 \xrightarrow{\text{inst}}_{d'} ... \xrightarrow{\text{inst}}_{d'} E_3.R.E_2.E_2^1...E_2^n$ and $\mathrm{fv}(E_2^n.z_n) = \{x_1, ..., x_m\}$.

By a similar argument to that in the base case $\{(x_1, l_1), ..., (x_m, l_m)\} \subseteq \mathrm{lhb}(E_3.R.E_2.E_2^1...E_2^{n-1})$.

Case split on where z_k is bound and prove $l_1, ..., l_m < \min(\mathrm{letlabels}(E_2^n))$.

➤*Case* $z_k \in E_3.R.E_2$ **:** The case split ensures that there exists E_3' and E_3'' such that

$$E_3.R.E_2 = E_3'.\mathbf{let}^{l_k}\, z_k = E_2^{k+1}.z_{k+1} \;\mathbf{in}\; E_3''$$

Because $E_3.R.E_2.z_0$ is closed, $\{x_1, ..., x_m\}$ are bound by the hole binders of E_3'. By assumption $E_3.R.E_2.z_0$ is the direct result of a labelling, so by the definition of $\mathrm{label}_{\_}(-)$ the labels of the binders in E_3', and thus the labels of the binders of $\{x_1, ..., x_m\}$, are all less than the labels in E_2^{k+1} as required.

➤*Case* $z_k \in E_2^1...E_2^{k-1}$ **:** WLoG assume $z_k \in E_2^i$ for some i, then $\mathrm{fv}(E_2^{k+1}) \subseteq \mathrm{fv}(E_2^i.z_i)$. By induction the free variables of $E_2^i.z_i$ are bound by labels less than the let labels of E_2^i which has as a subset the let labels of E_2^{k+1}.

❑

3.3.32 Lemma. *If* $E_3.R.E_2.z_0 = \mathrm{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0) \wedge E_3.R.E_3.z_0$ *closed then for all* $n \in \mathbb{N}$ *if* $E_3.R.E_2.z_0 \xrightarrow{inst}_{d'} ... \xrightarrow{inst}_{d'} E_3.R.E_2.E_2^1...E_2^n$ *then there exists* $\hat{E}_2, l$ *such that*

$$E_2^n = \mathrm{label}_l(\hat{E}_2)$$

Proof. We proceed by assuming $E_3.R.E_2.z_0 = \text{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0)$ and $E_3.R.E_3.z_0$ closed and proving the consequence by induction on n, the number of transitions.

➤*Case* $n = 0$ **:** $E_0 = \_$ and $\_ = \text{label}_l(\_)$ for any l.

➤*Case* $n = k + 1$ **:** Case split on where z_k is bound:

➤*Case* $z_k \in E_3.R.E_2$ **:** By assumption this context is the direct result of a labelling and so the subterm $E_2^{k+1}.z_{k+1}$ is also.

➤*Case* $z_k \in E_2^1...E_2^k$ **:** WLoG assume $z_k \in \text{hb}(E_2^i)$. By induction $E_2^i.z_i$ is the direct result of a labelling and so the subterm $E_2^{k+1}.z_{k+1}$ is also.

❑

We can now prove that every chain of contiguous instantiations has decreasing labels.

3.3.33 Lemma. *If* $E_3.R.E_2.z_0 = \text{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0) \wedge E_3.R.E_3.z_0$ *closed then for all* $n \in \mathbb{N}$ *if* $E_3.R.E_2.z_0 \xrightarrow{inst}_{d'} ... \xrightarrow{inst}_{d'} E_3.R.E_2.E_2^1...E_2^n$ *and* $\{(z_0, l_0), ..., (z_n, l_n)\} \subseteq$ *lhb*$(E_3.R.E_2.E_2^1...E_2^{n-1})$ *then*

$$l_1, ..., l_m < \min(\text{letlabels}(E_2^n))$$

Proof. Assume

$$E_3.R.E_2.z_0 = \text{label}_0(\hat{E}_3.\hat{R}.\hat{E}_2.z_0) \quad (3.60)$$

$$E_3.R.E_3.z_0 \text{ closed} E_3.R.E_2.z_0 \xrightarrow{\text{inst}}_{d'} ... \xrightarrow{\text{inst}}_{d'} E_3.R.E_2.E_2^1...E_2^{k+1} \quad (3.61)$$

$$\{(z_0, l_0), ..., (z_{k+1}, l_{k+1})\} \subseteq \text{lhb}(E_3.R.E_2.E_2^1...E_2^k) \quad (3.62)$$

We are required to prove $l_{k+1} < l_k$. We begin by case splitting on where z_k is bound.

➤*Case* $z_k \in \text{hb}(E_3.R.E_2)$ **:** In this case, for some E_3' and E_3'' we have

$$E_3.R.E_2 = E_3'.\textbf{let}^{l_k} z_k = E_2^{k+1}.z_{k+1} \textbf{ in } E_3''$$

By 3.60 this term is the direct result of a labelling. By 3.61 the term is closed so z_{k+1} must be bound in either E_2^{k+1} or E_3', but by the definition of $\mathrm{label}_{\_}(-)$ these contexts only have lets labelled less than l_k.

➤*Case* $z_k \in \mathrm{hb}(E_2^1...E_2^{k+1})$ **:** WLoG assume $z_k \in \mathrm{hb}(E_2^i)$ for some i so that there exists E_2' and E_2'' such that $E_2^i = E_2'.\mathbf{let}^{l_k} z_k = E_2^{k+1}.z_{k+1}$ **in** E_2'' and thus we have the term

$$E_3.R.E_2.E_2^1...E_2'.\mathbf{let}^{l_k} z_k = E_2^{k+1}.z_{k+1} \ \mathbf{in} \ E_2''.E_2^{i+1}...E_2^{k+1}.z_{k+1}$$

As reduction does not create free variables this term is closed and so z_{k+1} must be bound in either E_2^{k+1}, E_2' or $E_3.R.E_2.E_2^1...E_2^{i-1}$. Case split on these possibilities proving $l_{k+1} < l_k$ in each case.

➤*Case* E_2^{k+1} **:** By 3.3.32 this term is the direct result of a labelling. By the definition of $\mathrm{label}_{\_}(-)$ all lets in this context have smaller labels than l_k.

➤*Case* E_2' **:** E_2' is a subcontext of E_2^i and by 3.3.32 $E_2^i.z_i$ is the direct result of a labelling, thus E_2' is the direct result of a labelling. By the definition of $\mathrm{label}_{\_}(-)$ all lets in E_2' must have labels below l_k.

➤*Case* $E_3.R.E_2.E_2^1...E_2^{i-1}$ **:** In this case $z_k \in \mathrm{fv}(E_2^i.z_i)$ therefore by Lemma 3.3.31 $l_{k+1} < l_k$ as required.

❑

We can now show that all instantiate chains are finite.

3.3.34 Lemma. *For all closed a, there exists an a' such that $a \xrightarrow{insts}_{d'}^{*} a'$ and a'* $\mathsf{inf_d}$

Proof. If a can perform an instantiation reduction it is of the form $E_3.R.E_2.z_0$ and, as it is closed, $z_0 \in \mathrm{hb}(E_3.R.E_2)$. If z_0 instantiates to anything except $E_2'.z_1$ for some E_2' and z_1 then the term is in INF; we show this for the case where z_0 instantiates to $E_2'.(u_1, u_2)$. There are two possibilities for R, either $\_ \ u$ or $\pi_r \ \_$. Suppose the former, then

$$E3.R.E2.z0 = E3.(E2.E2'.(u1, u2)u)$$

which by unique decomposition (Lemma 2.2.1) is stuck and so in INF. So suppose the latter, then the term is not stuck as it matches the LHS of the projection rule. Again by unique decomposition we are in INF.

We now consider the only case in which we are not in INF after an instantiation, namely when we result in a term of the form $E_3.R.E_2.E_2'.z_1$. By Lemma 3.3.33 there is a well ordering on the lets that bind successively instantiated variables. Thus there can only be finitely many instantiations of this form before one of the proceeding cases holds. ❑

3.3.35 Lemma ($[\![-]\!]^-$ Invariant Under Insts). *If* $\mathrm{wf}[a] \ \wedge\ \Phi \blacktriangleleft a \ \wedge\ a \xrightarrow{insts}{}^{*}_{d'} a'$ *then* $[\![a]\!]^\Phi = [\![a']\!]^\Phi$

Proof. We first prove the single step case by induction on $a \xrightarrow{\text{insts}}_{d'} a'$:

➤*Case* (inst-1) : First note that by alpha conversion z and the hole binders of $E_3.R.E_2$ are all distinct.

$$\begin{aligned}
[\![\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.z]\!]^\Phi &= [\![z]\!]^{\Phi\,,\,z\mapsto[\![u]\!]^\Phi,\mathcal{E}_c[E_3.R.E_2]^\Phi} \\
&= [\Phi\ ,\ z \mapsto [\![u]\!]^\Phi, \mathcal{E}_c[E_3.R.E_2]^\Phi]^*(z) \\
&= [\![u]\!]^\Phi \\
&= [\![u]\!]^{\Phi\,,\,z\mapsto[\![u]\!]^\Phi,\mathcal{E}_c[E_3.R.E_2]^\Phi} && (3.63)\\
&= [\![u]\!]^{\Phi\,,\,z\mapsto[\![u]\!]^\Phi,\mathcal{E}_c[E_3.R.E_2]^\Phi\,,\,z\mapsto[\![u]\!]^\Phi} && (3.64)\\
&= [\![\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.u]\!]^\Phi
\end{aligned}$$

Where 3.63 is valid by part (ii) of Lemma 3.3.27 and 3.64 holds as $z \notin \mathrm{hb}(E_3.R.E_2)$.

➤*Case* (inst-2) : Similar to the previous case.

➤*Case* (instrec-1), (instrec-2) : $[\![z]\!]^{\Phi'} = \Phi'(z)$ which is equal to $[\![\lambda^z x.a]\!]^{\Phi'}$ for any environment that is well-formed w.r.t $\lambda^z x.a$.

➤*Case* (cong) : Assume $\mathrm{wf}[E_3.a]$ and $\Phi \blacktriangleleft E_3.a$. By Lemma 3.3.7 $\mathrm{wf}[a]$. Let $\Phi' = \Phi, \mathcal{E}_c[E_3]^\Phi$, then $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi')$. By induction $[\![a]\!]^{\Phi'} = [\![a']\!]^{\Phi'}$ (*). Now $[\![E_3.a]\!]^\Phi = [\![a]\!]^{\Phi'}$ and $[\![E_3.a']\!]^\Phi = [\![a']\!]^{\Phi'}$, thus by (*) we are done.

❑

3.3.36 Lemma ($[\![-]\!]^\Phi$ Source-Value Property). *if* $\mathrm{wf}[a] \ \wedge\ a\ \ \mathsf{inf}^\circ_\mathsf{d} \ \wedge\ \Phi \blacktriangleleft a \ \wedge\ [\![a]\!]^\Phi$ *cval then* a *d'val*

Proof. This proof is the same apart from the identifier case, which is immediate as identifiers are values. ❑

Notice in the next lemma that an extra restriction is needed when compared to the corresponding λ_r lemma, that is, the value u must not be an identifier.

3.3.37 Lemma ($[\![-]\!]$ Outer Value Preservation). *For all $\lambda_{d'}$ values u that are not identifiers:*

(a) If $\mathrm{wf}[u] \ \wedge \ \Phi \blacktriangleleft u \ \wedge \ [\![u]\!]^{\Phi} = \lambda x{:}\tau.e$ then there exists E_2, a, z and j such that $u = E_2.\lambda^j x{:}\tau.a$

(b) $[\![u]\!]^{\Phi} = (v_1, v_2) \implies \exists\, E_2, u_1, u_2.\ u = E_2.(u_1, u_2)$

3.3.38 Definition. Candidate bisimulation

$$R \triangleq \{(e, e') \mid \exists\, a.\ \mathrm{wf}[a] \ \wedge \ a \ \textsf{closed} \ \wedge \ e = [\![a]\!]^{\varnothing} \ \wedge \ e' = \epsilon[a]\}$$

❑

3.3.39 Lemma (c-d' Correspondence). *If a closed $\wedge$ $\mathrm{wf}[a]$ $\wedge$ $[\![a]\!]^{\varnothing} \longrightarrow_c e'$ then there exists a', a'' such that $a \xrightarrow{insts}{}^{*}_{d'} a'' \longrightarrow_{d'} a' \ \wedge \ a''$ $\textsf{inf}_\textsf{d}$ and either*

(i) $e' = [\![a']\!]^{\varnothing}$; or

(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![a']\!]^{\varnothing}$

Proof. We prove the generalised statement that if $\mathrm{wf}[a] \ \wedge \ a \ \ \textsf{inf}^{\circ}_{\textsf{d}} \ \wedge \ \Phi \blacktriangleleft a$ and $[\![a]\!]^{\Phi} \longrightarrow_c e'$ then there exists a' such that $a \longrightarrow_{d'} a'$ and one of the following hold

(i) $e' = [\![a']\!]^{\Phi}$; or

(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![a']\!]^{\varnothing}$

Most cases in this proof transfer directly because the lemmas used in the λ_r case still hold here. However, the $[\![-]\!]^{-}$ outer-value preservation property does not hold directly, instead we have an extra constraint that the value not be an identifier. This changes the projection and application case; we show just the former. Identifiers are trivial to deal with as they are not in open INF.

➤*Case* $\pi_r\ a$ **:** In the λ_r proof, this case is further decomposed by the possible reductions of $\pi_r\ a$. We have to alter the case where the projection occurs to show that a is not an identifier so that the $[\![-]\!]^{-}$ outer-value preservation result can be used. This is easily done as by assumption $(\pi_r\ a)$ $\textsf{inf}^{\circ}_{\textsf{d}}$, and if a was an identifier, say z, then this would not hold as z would be in a destruct position. ❑

3.3.40 Lemma (d'-d Correspondence).

$$a \ \textit{closed} \wedge \ \textit{wf}[a] \ \wedge \ a \xrightarrow{l}_{d'} a' \ \wedge \ l \neq \textit{zero} \implies \exists\, e'.\ \epsilon[a] \longrightarrow_d e' \ \wedge \ e' = \epsilon[a']$$

Proof. The proof is the same as the λ_r case. The (inst-1) and (inst-2) cases follow, as they did in the λ_r case, by the inst match property. ❑

3.3.41 Lemma (cd Eventually Weak Simulation). *R is an eventually weak simulation from λ_c to λ_d*

3.3.42 Lemma (d-d' Correspondence).

$$a \ \textit{closed} \wedge \textit{wf}[a] \wedge \epsilon[a] \longrightarrow_d e' \implies \exists\, a', a''.\ a \xrightarrow{\text{zeros}\ *}_{d'} a'' \longrightarrow_{d'} a' \wedge a'' \ \mathsf{znf_d} \wedge e' = \epsilon[a']$$

Proof. We prove the generalised statement:

$$\text{wf}[a] \ \wedge \ a \ \ \mathsf{znf}^{\circ}_{\mathsf{d}} \ \wedge \ \epsilon[a] \longrightarrow_d e' \implies \exists\, a'.\ a \longrightarrow_{d'} a' \ \wedge \ e' = \epsilon[a']$$

Most cases in this proof transfer directly because the lemmas used in the λ_r case still hold here. As the instantiation rules have changed, we need to reprove the $\textbf{let}_0 \ z = a_1 \ \textbf{in} \ a_2$, $a_1\ a_2$ and $\pi_r\ a$ cases:

➤*Case* $\pi_r\ a$ **:** In the λ_r proof this case is further decomposed by the possible reductions of the erased term. We have to add an extra case to this for the instantiation:

➤*Case* $\pi_r\epsilon[a] = \pi_r(E_2.\textbf{let}\ z = u \ \textbf{in}\ E_2'.z)$ **:** We can assume that a $\mathsf{znf_r}$ and wf$[a]$ and $\pi_r\epsilon[a] \longrightarrow_d \pi_r\ E_2.\textbf{let}\ z = u \ \textbf{in}\ E_2'.u$. We have to prove that there exists an a' such that $\pi_r\ a \longrightarrow_{d'} a'$ and $\pi_r(E_2.\textbf{let}\ z = u \ \textbf{in}\ E_2'.u) = \epsilon[a']$. By case split $\epsilon[a] = E_2.\textbf{let}\ z = u \ \textbf{in}\ E_2'.z$. By $\epsilon[-]$ Source Context (Lemma 3.3.21) for some $\hat{E}_2, \hat{a}$ we have $a = \hat{E}_2.\hat{a} \ \wedge \ \epsilon[\hat{E}_2] = E_2$, therefore $\epsilon[\hat{a}] = \textbf{let}\ z = u \ \textbf{in}\ E_2'.z$. By $\mathsf{znf}^{\circ}_{\mathsf{d}}$ preserved by E_3 stripping (Lemma 3.3.20) $\hat{a}$ $\mathsf{znf}^{\circ}_{\mathsf{d}}$. As $\hat{a}$ $\mathsf{znf_r}$ and it erases to a 'let', then $\hat{a}$ must be a $\textbf{let}_0$, as supposing that it is a $\textbf{let}_1$ leads to a contradiction about its ZNF property. Thus $\hat{a} = \textbf{let}_0 \ z = a_1 \ \textbf{in} \ a_2$, $\epsilon[a_1] = u$, $\epsilon[a_2] = E_2'.z$ and a_1 d'val by well-formedness. By $\epsilon[-]$ Source Context (Lemma 3.3.21) for some $\hat{E}_2{}', \hat{a}_2$ we have $a_2 = \hat{E}_2{}'.\hat{a}_2 \ \wedge \ \epsilon[\hat{E}_2{}'] = E_2'$. It follows that $\epsilon[\hat{a}_2] = z$ thus $\hat{a}_2 = z$. Moreover $a = \pi_r\ \hat{E}_2.\textbf{let}\ z = a_1 \ \textbf{in}\ \hat{E}_2{}'.z$ which reduces under $\lambda_{d'}$ to

$\pi_r\ \hat{E_2}.\mathbf{let}\ z = a_1\ \ \mathbf{in}\ \ \hat{E_2}'.a_1$. It is then simple enough to check that this erases to $\pi_r\ E_2.\mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_2'.u$.

➤*Case* $a_1\ a_2$ **:** Similar to the above proof.

➤*Case* $\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2$ **:** We have to consider the case where this term erases to a term that can do an instantiation:

➤*Case* $\mathbf{let}\ z\ =\ \epsilon[a_1]\ \mathbf{in}\ \epsilon[a_2]\ =\ \mathbf{let}\ z\ =\ u\ \ \mathbf{in}\ \ E_3.R.E_2.z$ **:** We can assume that $\mathsf{wf}[\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2]\ \wedge\ (\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ a_2)\ \mathsf{znf}^{\circ}_{\mathsf{d}}$ and $\mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.R.E_2.z \longrightarrow_d \mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.R.E_2.u$. By $\epsilon[-]$ Source Context (Lemma 3.3.21) there exists a λ_d context $\hat{E}_3$ and a $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a}$ and $\epsilon[\hat{E}_3] = E_3$, thus as erase distributes over contexts, $\epsilon[\hat{a}] = R.E_2.z$. We can see by inspection of $\epsilon[-]$ that if an erase results in an R context, then the input to erase must have been an R context, therefore let $\hat{a} = R.\hat{a}'$ for some $\hat{a}'$ then $\epsilon[R.\hat{a}'] = R.E_2.z$ and as erase distributes over contexts $\epsilon[\hat{a}'] = E_2.z$. By $\epsilon[-]$ Source Context (Lemma 3.3.21) there exists $\hat{E_2}$ and $\breve{a}$ such that $\hat{a}' = \hat{E_2}.\breve{a}$ and $\epsilon[\hat{E_2}] = E_2$ therefore $\epsilon[\breve{a}] = z$ forcing $\breve{a} = z$. Putting this all together $a_2 = \hat{E_3}.R.\hat{E_2}.z$, by well-formedness a_1 $\mathsf{d'val}$ and so $(\mathbf{let}\ z = a_1\ \ \mathbf{in}\ \ a_2) = (\mathbf{let}_0\ \ z = a_1\ \ \mathbf{in}\ \ \hat{E_3}.R.\hat{E_2}.z) \longrightarrow_{d'} \mathbf{let}_0\ \ z = u\ \ \mathbf{in}\ \ \hat{E_3}.R.\hat{E_2}.a_1$. More over it is easy to check that this last term erases to $\mathbf{let}\ z = u\ \ \mathbf{in}\ \ E_3.R.E_2.u$.

❑

3.3.43 Lemma (d'-c Correspondence). *If* a *closed and* $\mathsf{wf}[a]$ *and* $a \xrightarrow{l}_{d'} a'$ *and* $l \neq \mathit{insts}$ *then there exists an* e' *such that* $[\![\ a\ ]\!]^{\varnothing} \longrightarrow_c e'$ *and either:*

(i) $e' = [\![\ a'\ ]\!]^{\varnothing}$

(ii) there exists e'' *such that* $e' \longrightarrow_c e''$ *and* $e'' = [\![\ a'\ ]\!]^{\varnothing}$

❑

3.3.44 Lemma (dc Simulation). R *is a weak simulation from* λ_d *to* λ_c ❑

3.3.45 Lemma (Value Correspondence). *if* $\Phi = \Phi_k$ *where*

$$\begin{array}{lcl} \Phi_0 & = & \varnothing \\ \Phi_{n+1} & = & \Phi_n, x_{n+1} \mapsto [\![\ u_{n+1}\ ]\!]^{\Phi_n} \quad \textit{where}\ \mathit{fv}([\![\ u_{n+1}\ ]\!]^{\Phi_n}) = \varnothing \end{array}$$

and $\Phi \blacktriangleleft u$ *and* $\mathsf{wf}[u]$ *then* $S(\Phi)[\![\ \epsilon[u]\ ]\!]_{\mathsf{val}} = [\![\ u\ ]\!]^{\Phi}$

Proof. The proof follows the λ_r proof, but with an extra case necessary as variables can now be values. We give the extra case:

➤*Case* z : Under the assumptions $\Phi = \Phi_k$, $z \in \mathrm{dom}(\Phi)$ and $\mathrm{wf}[z]$ we are required to prove $S(\Phi)(z) = \Phi^*(z)$.

As $z \in \mathrm{dom}(\Phi)$, there exists $j \in 1 .. k-1$ such that

$$\begin{array}{rcl} S(\Phi)(z) & = & S(\Phi_j\ ,\ z \mapsto [\![\, u_{j+1} \,]\!]^{\Phi_j})(z) \\ & = & S(\Phi_j)[\![\, u_{j+1} \,]\!]^{\Phi_j} \end{array}$$

As $\mathrm{fv}([\![\, u_{j+1} \,]\!]^{\Phi_j}) = \varnothing$ then $S(\Phi_j)[\![\, u_{j+1} \,]\!]^{\Phi_j} = [\![\, u_{j+1} \,]\!]^{\Phi_j}$. Furthermore, as for all $v \in \mathrm{cod}(\Phi)$ it is the case that $\mathrm{fv}(v) = \varnothing$, we have that $\Phi^* = \Phi$. It follows that $[\![\, u_{j+1} \,]\!]^{\Phi_j} = \Phi(z) = \Phi^*(z)$ as required. ❑

The proof of the main theorem follows in the same way as in λ_r as the argument is purely based upon lemmas that have been reproved for the λ_d case, namely Lemmas 3.3.10, 3.3.41, 3.3.14, 3.3.14, 3.3.11, 3.3.34, 3.3.35, 3.3.18, 3.3.44, 3.3.28, 3.3.22, 3.3.19, 3.3.45 and 3.3.25.

3.4 Conclusion

This chapter proves that the semantics of the delayed instantiation calculi are consistent with the standard CBV semantics, in the sense that their contextual equivalence relations coincide, confirming that our usual CBV intuition and reasoning still hold. This was achieved separately for λ_r and λ_d, showing their reduction strategies to be equivalent to λ_c. The two proofs used the same technique: construct an eventually weak bisimulation relation that relates terms according to $[\![\, - \,]\!]^-$ and the additional property that whenever e is related to e' under R then e terminates if and only if e' terminates. To construct the relation R an annotated calculus was introduced that explicitly differentiated between which parts of the term are "environment" and which "program", and also between functions and recursive unrollings of a function. A relation (R) on λ terms was then formed from carefully crafted projections out of annotated λ terms, with the intent that it relate terms reduced under λ_c to those of a delayed instantiation calculus in the way described above.

4

Update Via Versioned Modules

In this chapter we use the delayed instantiation techniques developed in Chapter 2 to give a simple, statically typed language that supports the style of dynamic update found in Erlang [AVWW96, AV91], a dynamically typed language that supports DSU at the module-level. We present a formal *update calculus* that has the following characteristics:

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

2. *Flexibility*. We allow any module in the system to be updated, including changes to the types of its definitions, as long as the resulting program is type-correct. Furthermore, the timing of an update can be controlled by the programmer, based on the insertion of an `update` primitive. Finally, the effects of an update can be controlled by using appropriate variable syntax. In combination, these features allow us to model a range of systems, from those that allow updating at any time in arbitrary (but type-correct) ways, to those that have timing and/or update-content restrictions (e.g. [GKW97]).

3. *Practicality*. The calculus is informed by observing implementation experience [Hic01], and the most widely used DSU implementation, Erlang [AV91]. We show how the calculus can be used to model a number of realistic situations and updating strategies.

4.1 The update calculus

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

4.1.1 Syntax

Figure 4.1 shows the syntax of the language, which is basically a first-order, simply-typed, call-by-value lambda calculus with two extensions: (1) a simple module system with novel variable lookup rules, and (2) an `update` primitive that allows loading a new version of a module during program execution.

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

Expressions e are mostly standard, including pairs and pair projection π_r, function abstractions and applications, and `let` binders. To update a module with a new version, or insert a new module, we provide a primitive `update`, which allows a module to be loaded into the program during execution. To support staged transitions from old to new code, we allow flexible access to module components: to access the z component of a module named M, one can write either $M.z$, which will use the newest version of the module M, or $M^n.z$ (for some n), which will use version n of the code. This semantics is analogous to the prefixing semantics of Erlang, but is slightly more general. In particular, Erlang requires *all* references to an external module to invoke the newest version of the code; control is only possible when referencing bindings within the same module.

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

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

Figure 4.1: Update calculus syntax

$\boxed{P \xrightarrow{M^n=m} P'}$ (applying a module update to a program)

$$\text{(update)}\quad \frac{\begin{array}{l}\emptyset \vdash \texttt{modules}\ (ms \cup \{\texttt{module}\ M^n = m\})\texttt{in}\ E_1\ .() : S \\ n > \text{maxversion}(M, ms)\end{array}}{\begin{array}{l}\texttt{modules}\ ms\ \texttt{in}\ E_1\ .\ \texttt{update} \xrightarrow{M^n=m} \\ \texttt{modules}\ (ms \cup \{\texttt{module}\ M^n = m\})\texttt{in}\ E_1\ .()\end{array}}$$

$\boxed{P \longrightarrow P'}$ (internal computation step of a program)

(ver) $\texttt{modules}\ ms\ \texttt{in}\ E_1\ .(M^n.z) \longrightarrow \texttt{modules}\ ms\ \texttt{in}\ E_1\ .\ v$
where $ms = \{..,\texttt{module}\ M^n = \{..\ z{:}T = v\ ..\},..\}$

(unver) $\texttt{modules}\ ms\ \texttt{in}\ E_1\ .(M.z) \longrightarrow \texttt{modules}\ ms\ \texttt{in}\ E_1\ .\ v$
where $ms = \{..,\texttt{module}\ M^n = \{..\ z{:}T = v\ ..\},..\}$
and $n = \text{maxversion}(M, ms)$

(let) $E_2\ .\ E_1\ .\ \texttt{let}\ x{:}T = v\ \texttt{in}\ e \longrightarrow E_2\ .\ E_1\ .\{v/x\}e$

(proj) $E_2\ .\ E_1\ .\ \pi_r(v_1, v_2) \longrightarrow E_2\ .\ E_1\ .\ v_r$

(app) $E_2\ .\ E_1\ .(\lambda x{:}S.e)v \longrightarrow E_2\ .\ E_1\ .\{v/x\}e$

where

$\text{mods}(\{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}) = \{M_1^{n_1}, .., M_k^{n_k}\}$
$\text{maxversion}(M, ms) = \max\ \{n \mid M^n \in \text{mods}(ms)\}$

Figure 4.2: Update calculus reduction rules

4.1.2 Semantics

Figure 4.2 presents the dynamics of the calculus. We define a small-step reduction relation $P \longrightarrow P'$, using evaluation contexts E_1 for expressions and E_2 for programs. Context composition is denoted by ".", as in $E_2\ .\ E_1\ .\ e$. The rules for (let), (app) and (proj) are standard, while the remaining three rules describe accessing module bindings and updating module definitions.

So that module updates work as we would expect, module component identifiers are not resolved by substitution, as is the case with local bindings (c.f. the (let) and (app) rules), but instead by 'lookup', analogous to the redex-time semantics of Chapter 2. In particular, the (ver) rule will resolve the component identifier z from version n of

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

$$\begin{array}{l}
\mathrm{modsig}(\{z_1{:}T_1 = v_1 \ ...\ z_n{:}T_n = v_n\}) \ = \ \{z_1{:}T_1, .., z_n{:}T_n\} \\
\mathrm{modctx0}(\{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}) \ = \\
\qquad \{M_1^{n_1}{:}\mathrm{modsig}(m_1), .., M_k^{n_k}{:}\mathrm{modsig}(m_k)\} \\
\mathrm{modctx}(ms) \ = \\
\qquad \mathrm{modctx0}(ms) \cup \{M{:}\sigma \mid \exists\ n.M^n{:}\sigma \ \in\ \mathrm{modctx0}(ms) \wedge n = \mathrm{maxversion}(M, ms)\}
\end{array}$$

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

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

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

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

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

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

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

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

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

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

Figure 4.3: Update calculus typing rules

module M when the expression $M^n.z$ appears in redex position. Similarly, the (unver) rule handles the $M.z$ case, with the difference being that the most recent version of module M is used. This semantics is crucial for properly implementing updating.

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

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

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

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

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

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

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

The type system provides the necessary checks to ensure that loading a module does not result in a program that will reduce to a stuck state (one in which the expression is not a value and yet no reduction rule applies). Figure 4.3 shows the type system for our calculus. The rules for the judgement $\Sigma;\Gamma \vdash e{:}T$ are the standard ones for the simply typed lambda calculus, extended in the obvious way to deal with the typing of module components. The `update` command is statically uninteresting and types as `unit`, as this is the type of $()$, the value it becomes after an (update) transition. The other two judgements are more interesting. $\Sigma \vdash P{:}S$ types whole programs and handles most of the complexity in typing modules. We use two auxiliary functions modsig and modctx: modsig determines the interface of a module given its body, and, given a set of modules, modctx determines the partial function that maps versioned module names M^n to their signatures and also maps the unversioned module names M to the signature of the highest versioned module with the same name. modctx can thus be used to determine the module context in which the program (including the module bodies themselves) should be typed. The single rule defining the judgement ensures that the expression and every module body can be typed in this context; this means that the modules are allowed to be mutually recursive, as every module name is available in the typing of each module.

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

4.1.3 Discussion

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

The calculus addresses the run-time mechanisms involved in implementing updating (that is, loading new modules and allowing existing code or parts thereof to refer to them), but does not cover all the important software development issues of managing updatable code. In practice, we would expect compiler support for aiding the development process, for example [Hic01] where user programs could refer to the 'current' and 'previous' versions of a module, and the compiler would fill in the absolute version number.

As many past researchers have observed, the timing of an update is critical to assuring its validity [Gup94, Lee83, FS91, Hic01]. We chose to support *synchronous* updates by using the `update` primitive to dictate when an update can occur. This makes it easier to understand the state(s) of the program which an update is applied to than the alternative *asynchronous* approach, and so makes it easier to write correct updates.

Of equal importance is the need to control an update's effect. Which modules will 'notice' the new version? Can an old version and a new version coexist? Different systems answer these questions differently. Many systems allow multiple versions to coexist [AV91, FS91, Hic01, Dug01, App94, PHL97], while others prefer one version at a time [GKW97, Gup94]. Our use of module versions allows multiple generations of a module to exist simultaneously, and provides explicit control over which version of a module we are referring to, allowing us to delimit the effect of an update. As such, we can model a variety of updating situations.

Finally, we assert that updatable programs must be reliable: if the program crashes frequently, there is little need to update it dynamically! For this reason, we imposed a static (link-time) type system to ensure that if an update is accepted by the system, then the resulting program will be type-correct. In addition to improved reliability, we also believe that type-correct programs are easier to reason about.

4.2 Update Strategies

To illustrate the expressive power of the update facility in our calculus we show several strategies for completing an update to a simple server application. To simplify the examples we henceforth work with an extended version of the calculus containing conditionals, lists and two built-in functions for input and output:

$$e ::= \ldots \mid [] \mid e :: e \mid \texttt{hd}\ \ e \mid \texttt{tl}\ \ e \mid \texttt{if}\ \ e = e\ \ \texttt{then}\ \ e\ \ \texttt{else}\ \ e \mid \mathrm{input} \mid \mathrm{output}$$

the meaning of list and conditional being the obvious ones (for example, see [Pie02]). The I/O functions input and output have types $\mathsf{unit} \to \mathsf{int}$ and $\mathsf{int} \to \mathsf{unit}$ respectively, acting as an I/O channel for integers. Figure 4.4 shows a simple server application that collects integers, processes them, and then outputs a result. This process is repeated. Whenever the server receives the integer zero, it performs an upgrade by calling update.

The update we will demonstrate involves making the program collect pairs of integers (x, y).

4.2.1 Stop and Switch

This is the simplest strategy. We stop receiving data, process what we have and then continue using the new code. First notice that the `update` is within the collect loop. Upon the return of update, collect calls itself, therefore we should alter collect to return the list immediately:

$$\mathrm{collect} = \lambda(l{:}\mathsf{int\ list})\lambda x{:}\mathsf{int}.l$$

```
modules {
  module IServer⁰ = {
    process = λ(l:int list).λx:int.
      if  l = [] then  x  else IServer.process (tl  l) ((hd  l)   x),
    collect = λ(l:int list).λx:int.
      if  x = 0  then  l  else  let i = input() in
        let _ = if  i = 0  then update else () in
          IServer.collect (i :: l) (x − 1)
  },
  module Main⁰ = {
    loop = λu:unit.
      let l = IServer.collect [] 10000 in
      let x = IServer.process l 0 in
        Main.loop()
  }
}in
  Main.loop()
```

Figure 4.4: Example integer server

This is not the long-term behaviour we would like of collect though (we want it to collect integers), so we have a problem. One solution is to introduce another function $\text{collect}'$ and call this instead of collect, however this would involve changing all callers, something we would like to avoid. The solution is to split the update into two updates. First we replace the IServer module updating collect to

$$\text{collect} = \lambda(l{:}\mathsf{int\ list})\lambda x{:}\mathsf{int}.\mathtt{let}\ \_ = \mathtt{update}\ \mathtt{in}\ l$$

Then on the second update we re-update collect and apply the rest of the update. This involves changing both modules to be those shown in Figure 4.5.

4.2.2 Complete and Switch

In the previous example, the update may have disrupted the batching of input by cutting short one of the input sequences. We now present an alternative way of performing the update so that this does not happen. The strategy we present, dubbed *complete and*

```
modules {
  module IServer^0 = {
    process = λ(l:(int * int)list).λx:int.
      if l = [] then x else IServer.process (tl l) (π_1(hd l) * π_2(hd l)   x),
    collect = λ(l:(int * int)list).λx:int.
      if x = 0 then l else let i = (input(), input()) in
        let _ = if π_1 i = 0 then update else () in
          IServer.collect (i :: l) (x − 1)
  },
  module Main^0 = {
    loop = λu:unit.
      let l = IServer.collect [] 10000 in
      let x = IServer.process l 0 in
        Main.loop()
  }
}in
  Main.loop()
```

Figure 4.5: Example integer server update

switch, involves completing the collection of single integers in progress at the time of the update before going on to accept the pairs of integers. We again use two updates:

1. As we want the collection to complete, the first update simply schedules another update for when the collection is complete:

```
collect = λ(l:int list).λx:int.
        if x = 0 then let _ = update in l else let i = input() in
            IServer.collect (i :: l) (x − 1)
```

2. Upon reaching the update at the end of the recursion we perform the full update as in figure 4.5.

4.2.3 Convert and Switch

A third update strategy that we might like to encode is to start using the new code immediately, by converting our current state to be compatible with the new program. The calculus we present here, like current implementations, cannot encode this strategy, because of conditions placed on the program by its continuation (call stack). After performing an update within the collect function, the loop we return to in the main module will always be the old loop – the one that called collect. This expects a list of integers to be returned by collect while if we have converted the state and switched to the new code, collect will return a list of pairs.

The good thing is that such an update will not pass the type check. The downside is that intuitively it should. The main loop does not use the list it receives concretely, it merely passes it to the process function, which, if the program has been consistently updated, should be expecting a list of integers. Systems that we look at later will not have this problem.

4.3 Conclusions

In this chapter we presented the update calculus, a flexible and practical formal system for understanding the effects of dynamic software updating. The calculus is expressive enough to model Erlang-style update and the design decisions present in existing DSU systems.

While our language is type safe and has provided a useful model for discussing existing languages and update techniques, it has some serious short comings as a practical language for writing updatable programs. We briefly discuss these deficiencies, which we will return to in Part II.

Whole program typecheck at update-time At update time the system must re-typecheck the whole program to ensure that type consistency is maintained. Although this provides a flexible and expressive system in which arbitrary type changes to modules are permitted (the signatures of different versions of a module need not be related), it is difficult to implement and has implications for performance, because the runtime system must maintain dynamic type information.

Manual conversion of data If the type of data manipulated by a module is changed, then it is the programmer's responsibility to ensure all of the old data is correctly converted. This is quite a burden for the programmer and very difficult to get right, especially in the the presence of references where data may have to be found on the heap. A more declarative method where the data of a given type is automatically converted would be less error prone.

No updatability guarantee We give no guarantee that any particular part of the system will be updatable. It is the users problem to decide how available his program will be for update at runtime, which will be difficult to decide in any system of sufficient complexity.

Part II
Proteus

5 Proteus

In the previous chapter we considered an Erlang-style update at the module level that relied on a run-time type check. In this chapter we present Proteus, a general-purpose DSU formalism for C-like languages. Proteus supports fine-grained updates at the sub-module level of *named types*, ensuring the type safety of updates dynamically by examining how the program's continuation depends on their representation. Proteus programs consist of function, data and named type definitions. In the scope of a named type definition $\mathsf{t} = \tau$ the programmer can use the name t and representation type τ interchangeably (except at reference types) but the distinction lets us control updates. Dynamic updates can add new types and new definitions, and also provide replacements for existing ones, with replacement definitions possibly at changed types. Functions can be updated even while they are on the call-stack: the current version will continue (or be returned to), and the new version is activated on the next call. Permitting the update of active functions is important for making programs more available to dynamic updates [AV91, Hic01, BH00]. We also support updating function pointers.

When updating a named type t from its old representation τ to a new one τ', the user provides a *type transformer function* c with type $\tau \to \tau'$. This is used to convert existing t values in the program to the new representation. To ensure an intuitive semantics, we require that at no time can different parts of the program expect different representations of a type t; a concept we call *representation consistency*. The alternative would be to allow new and old definitions of a type t be valid simultaneously. Then, we could *copy* values when transforming them, where only new code sees the copies [Gup94, Hic01].

While this approach would be type safe, old and new code could manipulate different copies of the same data, which is likely to be disastrous in a language with side-effects.

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

To enforce con-freeness Proteus source programs, which we will refer to as $\mathrm{Proteus}^{\mathbf{src}}$, are automatically annotated with explicit *type coercions*: $\mathbf{abs}_{\mathrm{t}}\ e$ converts e to type t (assuming e has the proper concrete type τ), and $\mathbf{con}_{\mathrm{t}}\ e$ does the reverse at points where e of type t is used concretely. The target language of this translation is called $\mathrm{Proteus}^{\mathbf{con}}$. The explicit coercions in $\mathrm{Proteus}^{\mathbf{con}}$ allow us to dynamically analyse the active program during an update to some type t to check for the presence of coercions $\mathbf{con}_{\mathrm{t}}$. During this check we take into account that subsequent calls to updated functions will always be to their new versions. If any $\mathbf{con}_{\mathrm{t}}$ occurrences are discovered, then the update is rejected. In the next chapter we will give a way of approximating this property statically.

In what follows we define two core calculi—$\mathrm{Proteus}^{\mathbf{src}}$ and $\mathrm{Proteus}^{\mathbf{con}}$—that formalize our approach to dynamic software updating. In section 5.1 we present $\mathrm{Proteus}^{\mathbf{src}}$, the language used by programmers for writing updatable programs. Section 5.1.3 presents $\mathrm{Proteus}^{\mathbf{con}}$, an extension of $\mathrm{Proteus}^{\mathbf{src}}$ that makes the usage of named types manifest in programs by introducing *type coercions*; these are used to support the operational semantics of dynamic updating and to ensure that the process is type-safe by ensuring con-freeness.

5.1 Proteus Source Language

In this section we present $\mathrm{Proteus}^{\mathbf{src}}$, the language used by programmers for writing updatable programs. $\mathrm{Proteus}^{\mathbf{src}}$ models a type-safe, procedural language with second

Integers	$n \in \mathrm{Int}$
Internal names	$x, y, z \in \mathrm{IVar}$
External names	$\mathsf{x}, \mathsf{y}, \mathsf{z} \in \mathrm{XVar}$
Record labels	$\mathrm{l} \in \mathrm{Lab}$
References	$r \in \mathrm{Ref}$
Type names	$\mathsf{t}, \mathsf{s} \in \mathrm{NT}$
Variables	$\mathrm{Var} = \mathrm{IVar} \uplus \mathrm{XVar} \uplus \mathrm{NT}$

Types	$\tau \in \mathrm{Typ}$	$::=$	$\mathsf{t} \mid \mathsf{int} \mid \mathsf{unit} \mid \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \mid \tau\ \mathbf{ref}$	
		$\mid$	$\tau_1 \rightarrow \tau_2$	
Expressions	$e \in \mathrm{Exp}$	$::=$	n	integers
		$\mid$	x	variables
		$\mid$	z	external names
		$\mid$	r	heap reference
		$\mid$	$\{\mathrm{l}_1 = e_1, \ldots, \mathrm{l}_n = e_n\}$	records
		$\mid$	$e.\mathrm{l}$	projection
		$\mid$	$e_1\ e_2$	application
		$\mid$	$\mathbf{let}\ z : \tau = e_1\ \mathbf{in}\ e_2$	let bindings
		$\mid$	$\mathbf{ref}\ e$	ref. creation
		$\mid$	$e_1 := e_2$	assignment
		$\mid$	$!e$	dereference
		$\mid$	$\mathbf{if}\ e_1 = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4$	conditional
		$\mid$	$\mathbf{update}$	dynamic update
Values	$v \in \mathrm{Val}$	$::=$	$\mathsf{z} \mid n \mid \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\} \mid r$	
Programs	$P \in \mathrm{Prog}$	$::=$	$\mathbf{var}\ \mathsf{z} : \tau = v\ \mathbf{in}\ P$	
		$\mid$	$\mathbf{fun}\ \mathsf{z}_1(x : \tau_1) : \tau_1' = e_1 \ldots$	
			$\ldots \mathbf{and}\ \mathsf{z}_n(x : \tau_n) : \tau_n' = e_n\ \mathbf{in}\ P$	
		$\mid$	$\mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P$	
		$\mid$	e	

Figure 5.1: Syntax for Proteus$^{\mathbf{src}}$

class functions and mutable state (C-like) augmented with dynamic updating; its syntax is shown in Figure 5.1. Programs P are a series of top-level definitions followed by an expression e. A **fun** z... defines a top-level recursive function, and **var** z : τ... defines a top-level *mutable* variable (i.e., it has type τ **ref**). We allow mutually-recursive blocks of function definitions using the keyword **and**. A **type** t $= \tau \ldots$ defines the type t. Top-level variables z (a.k.a. *external names*) must be unique within P, and are not subject to α-conversion, so they can be unambiguously updated at run-time. Expressions e are largely standard. We abuse notation and write multi-argument functions

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

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

The **update** expression allows the program to accept a dynamic update, but before we discuss the semantics of this expression, let us consider an example program and update.

5.1.1 Example

Figure 5.2 shows a simple kernel for handling read and write requests on files or sockets, which one might want to dynamically update. Some functions and type definitions have been elided for simplicity and we use simple enumerated type definitions (e.g. fdtype) for clarity on the understanding that they could easily be encoded as integers. Reading from the bottom, the function loop is an infinite loop that repeatedly gets req objects (e.g., system calls) and then dispatches them to an appropriate handler using the dispatch function. This function first calls decode to determine whether a given file descriptor is a network channel or an open file (e.g., by looking in a table). If it is a network channel, dispatch calls getsock to get a sock object based on the given file descriptor (e.g., indexed in an array). Next, it decodes the remaining portion of the req to acquire the transmission flags. Finally, it finds an appropriate sockhandler object to handle the request and calls that handler. Handlers are needed to support different kinds of network channel, e.g., for datagram communications (UDP), for streaming connections (TCP), etc. Different handlers are implemented for each kind, and getsockhandler chooses the right one. A similar set of operations and types would be in place for files. After dispatch completes, its result is posted to the user, and the loop continues.

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

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

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

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

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

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

loop 0
```

Figure 5.2: A simple kernel for files and socket I/O

5.1.2 Update

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

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

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

- New type, function, and variable definitions.

Figure 5.3 shows dispatch from Figure 5.2 with some added **update** expressions. Here we can assume that the update expressions are inserted by the programmer, but later on in Section 6.4.1 we will consider how these can be inserted automatically.

The Figure also shows explicit type coercions which will help us to determine if the update is safe. We discuss these in the next section.

5.1.3 Proteus Target Language

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

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

Given a type definition $\mathbf{type}\ \mathsf{t} = \tau$, the $\mathrm{Proteus}^{\mathbf{src}}$ typing rules effectively allow values of type t and type τ to be used interchangeably, as is typical.

In Figure 5.2, within the scope of a type definition like $\mathbf{type}\ \mathsf{sockhandler} = \ldots$ the type sockhandler is a synonym for its definition. For example, the expression $h\ (k,\ s.\mathrm{buf},\ \mathit{flags})$ in dispatch uses h, which has type sockhandler, as a function. In this case, we say that the named type sockhandler is being used *concretely*. However, there are also parts of the program that treat data of named type *abstractly*, i.e., they do not rely on its representation. For example, the getsockhandler function simply returns a sockhandler value; that the value is a function is incidental.

In $\mathrm{Proteus}^{\mathbf{con}}$ (but not in the user source language, $\mathrm{Proteus}^{\mathbf{src}}$) all uses of a named type definition $\mathsf{t} = \tau$ are made explicit with type coercions: $\mathbf{abs}_{\mathsf{t}}\ e$ converts e to type t (assuming e has the proper type τ), and $\mathbf{con}_{\mathsf{t}}\ e$ does the reverse. Figure 5.3 illustrates the dispatch function from Figure 5.2 with these coercions inserted. As examples, we can see that a handResult value is constructed in the last line from -1 via the coercion $(\mathbf{abs}_{\mathsf{handResult}}\ -1)$. Conversely, to invoke h (in the expression for res), it must be converted from type sockhandler via the coercion $(\mathbf{con}_{\mathsf{sockhandler}}\ \mathrm{h})\ (\ldots)$.

Type coercions serve two purposes operationally. First, they are used to prevent updates to some type t from occurring at a time when existing code still relies on the old representation. In particular, the presence of $\mathbf{con}_{\mathsf{t}}$ clearly indicates where the concrete representation of t is relied upon, and therefore can be used as part of a static or dynamic analysis to avoid an invalid update (§5.5).

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

Figure 5.3: dispatch with explicit **update** and coercions

Second, coercions are used to "tag" abstract data so it can be converted to a new representation should its type be updated. In particular, all expressions of type t occurring in the program will have the form $\mathbf{abs}_{\mathsf{t}}\ e$. Therefore, given a user-provided transformer function c_{t} which converts from the old representation of t to the new, we can rewrite each instance at update-time to be $\mathbf{abs}_{\mathsf{t}}\ (c_{\mathsf{t}}\ e)$. This leads to a natural CBV evaluation strategy for transformers in conjunction with the rest of the program (§5.4).

The typing rules for coercions are simple:

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

In fact the target type system shown in Figures 5.4, 5.5 and 5.6 is simple, although we have to become more complex to insert coercions automatically.

5.2 Automatic coercion insertion

Compiling a Proteus$^{\mathbf{src}}$ program to a Proteus$^{\mathbf{con}}$ program requires inserting type coercions to make explicit the implicit uses of the type equality in the source program. Our methodology is based on coercive subtyping [BTCGS91]. As is usual for coercive subtyping systems, we define our translation inductively over source language typing derivations. In particular, we define a judgement $\Gamma \vdash P : \tau \rightsquigarrow P'$ by which a Proteus$^{\mathbf{src}}$ program P is translated to Proteus$^{\mathbf{con}}$ program P'. (The typing environment Γ is a finite mapping from variables to types and from type names to types. As is usual we will sometimes write a mapping using a list notation, e.g., $\Gamma \equiv x : \tau, \mathsf{t} = \tau'$.) Our

Expression Typing: $\Gamma \vdash e : \tau$

$$\Gamma \vdash n : \mathsf{int} \qquad \text{(EXPR.INT)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(EXPR.VAR)}$$

$$\frac{\Gamma(\mathsf{z}) = \tau}{\Gamma \vdash \mathsf{z} : \tau} \qquad \text{(EXPR.XVAR)}$$

$$\frac{\Gamma(r) = \tau\ \mathbf{ref}}{\Gamma \vdash r : \tau\ \mathbf{ref}} \qquad \text{(EXPR.LOC)}$$

$$\frac{\Gamma \vdash e_{i+1} : \tau_{i+1} \qquad i \in 1..(n-1) \qquad n \geq 0}{\Gamma \vdash \{\mathrm{l}_1 = e_1, \ldots, 1_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}} \qquad \text{(EXPR.RECORD)}$$

$$\frac{\Gamma \vdash e : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}}{\Gamma \vdash e.\mathrm{l}_i : \tau_i} \qquad \text{(EXPR.PROJ)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \text{(EXPR.APPU)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e : \tau & \Gamma \vdash e' : \tau \\ \Gamma \vdash e_1 : \tau' & \Gamma \vdash e_2 : \tau'\end{array}}{\Gamma \vdash \mathbf{if}\ e = e'\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'} \qquad \text{(EXPR.IF)}$$

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}\ e : \tau\ \mathbf{ref}} \qquad \text{(EXPR.REF)}$$

Figure 5.4: Expression typing rules for Proteus$^{\mathbf{con}}$, the target of the translation (Part I)

$\boxed{\textbf{Expression Typing: } \Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau\ \mathbf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \text{(EXPR.DEREF)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau\ \mathbf{ref} \\ \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \qquad \text{(EXPR.ASSIGN)}$$

$$\Gamma \vdash \mathbf{update} : \mathsf{int} \qquad \text{(EXPR.UPDATE)}$$

$$\frac{\Gamma \vdash e : \mathsf{t} \qquad \Gamma(\mathsf{t}) = \tau}{\Gamma \vdash \mathbf{con}_{\mathsf{t}}\ e : \tau} \qquad \text{(EXPR.CON)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(\mathsf{t}) = \tau}{\Gamma \vdash \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}} \qquad \text{(EXPR.ABS)}$$

Figure 5.5: Expression typing rules for Proteus$^{\mathbf{con}}$, the target of the translation (Part II)

primary aim is that the translation be *deterministic*, so that *where* coercions are inserted is intuitive to the programmer. Secondarily, we wish the resulting Proteus$^{\mathbf{con}}$ program to be *efficient*, with a minimal number of inserted coercions and other computational machinery.

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

Abstraction and Concretion as Subtyping

To properly insert $\mathbf{con}_{\mathsf{t}}$ and $\mathbf{abs}_{\mathsf{t}}$ coercions in the source program P, we must identify where P uses values of type t concretely and abstractly. We do this by defining a mostly-standard subtyping judgement for Proteus$^{\mathbf{src}}$, written $\Gamma \vdash \tau <: \tau'$, with two key rules.

$\boxed{\textbf{Program Typing: } \Gamma \vdash e : \tau}$

$$\frac{\begin{array}{l}\Gamma, \mathsf{t} = \tau' \vdash_P P : \tau \\ \Gamma \vdash \tau'\end{array}}{\Gamma \vdash_P \textbf{type } \mathsf{t} = \tau' \textbf{ in } P : \tau} \quad (\text{PROG.TYPE})$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \mathsf{z}_1 : \tau_1 \overset{\mu_1;\Delta'_1}{\longrightarrow} \tau'_1, \ldots, \mathsf{z}_n : \tau_1 \overset{\mu_n;\Delta'_n}{\longrightarrow} \tau'_n \\ \Gamma', x : \tau_i \vdash e_i : \tau_i \qquad i \in 1..n \qquad \Gamma' \vdash_P P : \tau\end{array}}{\Gamma \vdash_P \begin{array}{l}\textbf{fun } \mathsf{z}_1(x : \tau_1) : \tau'_1 = e_1 \ldots \\ \quad \textbf{and } \mathsf{z}_n(x : \tau_n) : \tau'_n = e_n \textbf{ in } P : \tau\end{array}} \quad (\text{PROG.FUN})$$

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma, \mathsf{z} : \tau' \textbf{ ref} \vdash_P P : \tau}{\Gamma \vdash_P \textbf{var } \mathsf{z} : \tau' = v \textbf{ in } P : \tau} \quad (\text{PROG.VAR})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_P e : \tau} \quad (\text{PROG.EXPR})$$

Figure 5.6: Program typing rules for Proteus$^{\textbf{con}}$, the target of the translation

First, given a value of type τ it can be *abstracted* as having type t under the assumption $\mathsf{t} = \tau$:

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

Conversely a value of type t can be treated *concretely* as having type τ:

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

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

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

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

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

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

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

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

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

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

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

Making Subtyping Algorithmic

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

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

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

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

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

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

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

$$\dfrac{}{\Gamma \vdash \tau <: \tau}\ (\textsc{refl})$$

$$\dfrac{\Gamma, \mathsf{t} = \tau \vdash \tau <: \tau'}{\Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \tau'}\ (\textsc{con}) \qquad \dfrac{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \tau}{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \mathsf{t}}\ (\textsc{abs})$$

$$\dfrac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash \tau_2 <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ (\textsc{fun})$$

$$\dfrac{\Gamma \vdash \tau_1 <: \tau_1' \quad \cdots \quad \Gamma \vdash \tau_k <: \tau_k' \quad k \leq n}{\Gamma \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\} <: \{l_1 : \tau_1', \ldots, l_k : \tau_k\}}\ (\textsc{rec})$$

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

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

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

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

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

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau <: \tau'}{\Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \tau'} \ (\textsc{con-c})$$

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau' <: \tau}{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \mathsf{t}} \ (\textsc{abs-c})$$

$$\frac{\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \qquad \mathit{Height}(\mathsf{t}, \Gamma) \geq \mathit{Height}(\mathsf{s}, \Gamma) \qquad \Gamma \vdash \tau <: \mathsf{s}}{\Gamma \vdash \mathsf{t} <: \mathsf{s}} \ (\textsc{con-n})$$

$$\frac{\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \qquad \mathit{Height}(\mathsf{s}, \Gamma) > \mathit{Height}(\mathsf{t}, \Gamma) \qquad \Gamma \vdash \mathsf{t} <: \tau'}{\Gamma \vdash \mathsf{t} <: \mathsf{s}} \ (\textsc{abs-n})$$

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

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

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

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

We can use this algorithmic subtyping relation to generate coercion contexts as shown in Figure 5.7. As an example, we show how the derivation of $\mathsf{u} <: \mathsf{v}$ from earlier would be translated (assuming that $\Gamma \equiv \mathsf{t} = \mathsf{int}, \mathsf{s} = \mathsf{t}, \mathsf{u} = \mathsf{s}, \mathsf{v} = \mathsf{t}$).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{t} <: \mathsf{t} \rightsquigarrow \_ \qquad \mathit{Height}(\mathsf{v},\Gamma) > \mathit{Height}(\mathsf{t},\Gamma)}\ (\textsc{t.coer.refl})}{\Gamma \vdash \mathsf{t} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \_ \qquad \mathit{Height}(\mathsf{s},\Gamma) \geq \mathit{Height}(\mathsf{v},\Gamma)}\ (\textsc{t.coer.abs-n})}{\Gamma \vdash \mathsf{s} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \mathbf{con}_{\mathsf{s}}\ \_ \qquad \mathit{Height}(\mathsf{u},\Gamma) \geq \mathit{Height}(\mathsf{v},\Gamma)}\ (\textsc{t.coer.con-n})}{\Gamma \vdash \mathsf{u} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \mathbf{con}_{\mathsf{s}}\ \mathbf{con}_{\mathsf{u}}\ \_}\ (\textsc{t.coer.con-n})$$

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

5.2.1 Theorem. *We refer to the subtyping relation with* (CON) *and* (ABS) *as* $\vdash^{nd}$ *and the relation that replaces these with* (CON-N), (ABS-N), (CON-C), (ABS-C) *as* $\vdash^{alg}$*. Then* $\Gamma \vdash^{nd} \tau <: \tau'$ *if and only if* $\Gamma \vdash^{alg} \tau <: \tau'$*.*

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

Algorithmic expression typing

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

$\boxed{\Gamma \vdash \tau <: \tau' \rightsquigarrow \mathcal{C}}$

$$\frac{}{\Gamma \vdash \tau <: \tau \rightsquigarrow \_} \quad \text{(T.COER.REFL)}$$

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau <: \tau' \rightsquigarrow \mathcal{C}}{\Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \tau' \rightsquigarrow \mathcal{C}[\mathbf{con}_{\mathsf{t}}\ \_]} \quad \text{(T.COER.CON-C)}$$

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau' <: \tau \rightsquigarrow \mathcal{C}}{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \mathsf{t} \rightsquigarrow (\mathbf{abs}_{\mathsf{t}}\ \_)[\mathcal{C}]} \quad \text{(T.COER.ABS-C)}$$

$$\frac{\begin{array}{l}\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \\ \mathit{Height}(\mathsf{t}, \Gamma) \geq \mathit{Height}(\mathsf{s}, \Gamma) \qquad \Gamma \vdash \tau <: \mathsf{s} \rightsquigarrow \mathcal{C}\end{array}}{\Gamma \vdash \mathsf{t} <: \mathsf{s} \rightsquigarrow \mathcal{C}[\mathbf{con}_{\mathsf{t}}\ \_]} \quad \text{(T.COER.CON-N)}$$

$$\frac{\begin{array}{l}\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \\ \mathit{Height}(\mathsf{s}, \Gamma) > \mathit{Height}(\mathsf{t}, \Gamma) \qquad \Gamma \vdash \mathsf{t} <: \tau' \rightsquigarrow \mathcal{C}\end{array}}{\Gamma \vdash \mathsf{t} <: \mathsf{s} \rightsquigarrow (\mathbf{abs}_{\mathsf{t}}\ \_)[\mathcal{C}]} \quad \text{(T.COER.ABS-N)}$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \rightsquigarrow \mathcal{C}_1 \qquad \Gamma \vdash \tau_2 <: \tau_2' \rightsquigarrow \mathcal{C}_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2' \rightsquigarrow \lambda(f : \tau_1 \rightarrow \tau_2).\lambda(x : \tau_1').\mathcal{C}_2[f\ (\mathcal{C}_1[x])]\ \ \_} \quad \text{(T.COER.FUN)}$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_1' \rightsquigarrow \mathcal{C}_1 \quad \cdots \quad \Gamma \vdash \tau_k <: \tau_k' \rightsquigarrow \mathcal{C}_2 \quad k \leq n}{\begin{array}{l}\Gamma \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\} <: \{l_1 : \tau_1', \ldots, l_k : \tau_k\} \rightsquigarrow \\ \mathbf{let}\ x : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} = \_\ \mathbf{in}\ \{\mathrm{l}_1 = \mathcal{C}_1[x.\mathrm{l}_1], \ldots \mathrm{l}_n = \mathcal{C}_n[x.\mathrm{l}_n]\}\end{array}} \quad \text{(T.COER.REC)}$$

Figure 5.7: Coercion generation via the subtyping relation

would not be algorithmic, we only require *unfolding*, a weaker form of subsumption, at e_1.[1] To this end, we introduce an unfolding judgement $\Gamma \vdash \mathsf{t} \lhd \tau$, that relates t and τ if the definition $\mathsf{t} = \tau$ holds directly or transitively, inserting an explicit concretion every time it unfolds a named type to its definition. The unfolding judgement is then used whenever a specific type is required in the typing judgement, i.e., in the application, dereference, assignment, and projection rules. This relation is defined as follows:

[1]The alert reader may have noticed that the subtype relation is in fact sufficient to get algorithmic behaviour at application, provided we drop the notion of applying subsumption to the arguments of functions, and instead apply it to the *function type*. Although theoretically simpler, in practice we want to avoid coercing functions, which is expensive. Moreover, we need the unfolding at projection and dereference in any case, thus it seems a more general concept to apply unfolding at *destruct positions* — points where the top-level structure of a value is deconstructed [BHS$^+$03a].

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

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

The complete rules for the translation are given in Figures 5.7, 5.8 and 5.9. (The normal typing rules for $\mathrm{Proteus}^{\mathbf{src}}$ can be read from these figures by simply ignoring the $\rightsquigarrow\ \mathcal{C}$ parts.)

5.3 Specifying Dynamic Updates

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

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

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

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

- AB (Added Bindings) is a map from top-level identifiers z to pairs of types and binding values. These are used to specify new **fun** and **var** definitions.

As an example, say we wish to modify socket handling in Figure 5.2 to include a *cookie* argument for tracking security information (this was done at one point in Linux). This requires four changes: (1) we modify the definition of sockhandler to add the additional argument; (2) we modify the sock type to add new information (such as a destination address for which the cookie is relevant); (3) we modify existing handlers,

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

$$\Gamma \vdash n : \mathsf{int} \leadsto n \qquad \text{(T.EXPR.INT)}$$

$$\Gamma, x{:}\,\tau \vdash x : \tau \leadsto x \qquad \text{(T.EXPR.VAR)}$$

$$\Gamma, \mathsf{z}{:}\,\tau \vdash \mathsf{z} : \tau \leadsto \mathsf{z} \qquad \text{(T.EXPR.XVAR)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \leadsto e'_i \qquad (i \in 1..n)}{\begin{array}{l}\Gamma \vdash \{\mathrm{l}_1 = e_1, \ldots, \mathrm{l}_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \\ \leadsto \{\mathrm{l}_1 = e'_1, \ldots, \mathrm{l}_n = e'_n\}\end{array}} \qquad \text{(T.EXPR.REC)}$$

$$\frac{\Gamma \vdash e : \tau \leadsto e' \qquad \Gamma \vdash \tau \lhd \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \leadsto \mathcal{C}}{\Gamma \vdash e.\mathrm{l}_i : \tau_i \leadsto \mathcal{C}[e'].\mathrm{l}_i} \qquad \text{(T.EXPR.PROJ)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau \leadsto e'_1 & \Gamma \vdash e_2 : \tau'_1 \leadsto e'_2 \\ \Gamma \vdash \tau'_1 <: \tau_1 \leadsto \mathcal{C}_1 & \Gamma \vdash \tau \lhd \tau_1 \rightarrow \tau_2 \leadsto \mathcal{C}_2\end{array}}{\Gamma \vdash e_1\, e_2 : \tau_2 \leadsto \mathcal{C}_2[e'_1]\, \mathcal{C}_1[e'_2]} \qquad \text{(T.EXPR.APP)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau'_1 \leadsto e'_1 \\ \Gamma \vdash \tau'_1 <: \tau_1 \leadsto \mathcal{C} \\ \Gamma, x{:}\,\tau_1 \vdash e_2 : \tau_2 \leadsto e'_2\end{array}}{\Gamma \vdash \mathbf{let}\ x : \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau_2 \leadsto \mathbf{let}\ x : \tau_1 = \mathcal{C}[e'_1]\ \mathbf{in}\ e'_2} \qquad \text{(T.EXPR.LET)}$$

$$\frac{\Gamma \vdash e : \tau \leadsto e'}{\Gamma \vdash \mathbf{ref}\ e : \tau\ \mathbf{ref} \leadsto \mathbf{ref}\ e'} \qquad \text{(T.EXPR.REF)}$$

$$\frac{\Gamma \vdash e : \tau\ \mathbf{ref} \leadsto e'}{\Gamma \vdash\ !e : \tau \leadsto\ !e'} \qquad \text{(T.EXPR.DEREF)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau\ \mathbf{ref} \leadsto e'_1 \\ \Gamma \vdash e_2 : \tau' \leadsto e'_2 \\ \Gamma \vdash \tau' <: \tau \leadsto \mathcal{C}\end{array}}{\Gamma \vdash e_1 := e_2 : \mathsf{unit} \leadsto e'_1 := \mathcal{C}[e'_2]} \qquad \text{(T.EXPR.ASSIGN)}$$

$$\Gamma \vdash \mathbf{update} : \mathsf{int} \leadsto \mathbf{update} \qquad \text{(T.EXPR.UPDATE)}$$

Figure 5.8: Con/abs insertion for compiling $\mathrm{Proteus}^{\mathbf{src}}$ expressions

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

$$\frac{\begin{array}{l}\Gamma \vdash \tau \\ \Gamma, \mathsf{t} = \tau \vdash_P P : \tau \rightsquigarrow P'\end{array}}{\Gamma \vdash_P \textbf{type}\ \mathsf{t} = \tau\ \textbf{in}\ P : \tau \rightsquigarrow \textbf{type}\ \mathsf{t} = \tau\ \textbf{in}\ P'} \quad \text{(T.PROG.TYPE)}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \mathsf{z_i} : \tau_i \rightarrow \tau_i' \\ \Gamma', x : \tau_i \vdash e_i : \tau_i'' \rightsquigarrow e' \\ \Gamma' \vdash \tau_i'' <: \tau_i' \rightsquigarrow \mathcal{C} \\ \Gamma' \vdash_P P : \tau \rightsquigarrow P' \\ i \in 1..n\end{array}}{\Gamma \vdash_P \begin{array}{l}\textbf{fun}\ \mathsf{z_1}(x : \tau_1) : \tau_1' = e_1\ \textbf{and}\ \ldots \\ \textbf{fun}\ \mathsf{z_n}(x : \tau_n) : \tau_n' = e_n\ \textbf{in}\ P : \tau \rightsquigarrow \\ \textbf{fun}\ \mathsf{z_1}(x : \tau_1) : \tau_1' = \mathcal{C}[e_1]\textbf{and}\ \ldots \\ \textbf{fun}\ \mathsf{z_n}(x : \tau_n) : \tau_n' = \mathcal{C}[e_n]\ \textbf{in}\ P'\end{array}} \quad \text{(T.PROG.FUN)}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : \tau'' \rightsquigarrow e' \\ \Gamma \vdash \tau'' <: \tau' \rightsquigarrow \mathcal{C} \\ \Gamma, \mathsf{z} : \tau'\ \textbf{ref} \vdash_P P : \tau \rightsquigarrow P'\end{array}}{\Gamma \vdash_P \textbf{var}\ \mathsf{z} : \tau' = v\ \textbf{in}\ P : \tau \rightsquigarrow \textbf{var}\ \mathsf{z} : \tau' = \mathcal{C}[e']\ \textbf{in}\ P'} \quad \text{(T.PROG.VAR)}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e'}{\Gamma \vdash_P e : \tau \rightsquigarrow e'} \quad \text{(T.PROG.EXPR)}$$

Figure 5.9: Con/abs insertion for compiling Proteus$^{\mathbf{src}}$ programs

$\boxed{\textbf{Type well-formedness: } \Gamma \vdash \tau}$

$$\Gamma \vdash \mathsf{int} \qquad \text{(A.TYPE.WF.INT)}$$

$$\frac{\Gamma \vdash \tau_i \qquad i \in 1..n}{\Gamma \vdash \{\mathrm{l}_1 : \tau_1, ..., \mathrm{l}_n : \tau_n\}} \qquad \text{(A.TYPE.WF.RECORD)}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau\ \mathbf{ref}} \qquad \text{(A.TYPE.WF.REF)}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2} \qquad \text{(A.TYPE.WF.FUN)}$$

$$\frac{\mathsf{t} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{t}} \qquad \text{(A.TYPE.WF.NT)}$$

Figure 5.10: Type well-formedness

like udp_read, to add the new functionality, and (4) we modify the dispatch routine to call the handler with the new argument. The user must provide functions to convert existing sock and sockhandler objects.

The update is shown in Figure 5.11. The UN component specifies the new definitions of sock and sockhandler, along with type transformer functions sockh_coer and sock_coer, which are defined in AB. The AN component defines the new type cookie = int, and that it should be inserted above the definition of sockhandler (which refers to it). Next, UB specifies a replacement dispatch function that calls the socket handler with the extra security cookie, which is acquired by calling a new function security_info.

The AB component specifies the definitions to add. First, it specifies new handler functions udp_read$'$ and udp_write$'$ to be used in place of the existing udp_read and udp_write functions. The reason they are defined here, and not in UB, is that the new versions of these functions have a different type than the old versions (they take an additional argument). So that code will properly call the new versions from now on, the sock_coer maps between the old ones and the new ones. Thus, existing datastructures

$$
\begin{array}{lll}
\mathrm{UN}: & \mathsf{sockhandler} & \mapsto \\
 & \multicolumn{2}{l}{\quad (\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}, \mathrm{cookie} : \mathsf{cookie}\} \rightarrow \mathsf{int},\ \mathsf{sockh\_coer})} \\
 & \mathsf{sock} & \mapsto \quad (\{\mathrm{daddr} : \mathsf{int}, \ldots\}, \mathsf{sock\_coer}) \\
\mathrm{AN}: & \mathsf{sockhandler} & \mapsto \quad (\mathsf{cookie}, \mathsf{int}) \\
\mathrm{UB}: & \mathsf{dispatch} & \mapsto \\
 & \multicolumn{2}{l}{\quad (\mathsf{req} \rightarrow \mathsf{handResult},} \\
 & \multicolumn{2}{l}{\qquad \lambda(s)\ldots (\mathbf{con}_{\mathsf{sockhandler}}\ \mathrm{h})(\mathrm{k},\ (\mathbf{con}_{\mathsf{req}}\ \mathrm{s}).\mathrm{buf},\ \mathrm{flags},\ (\mathsf{security\_info}\ ())\ldots)} \\
\mathrm{AB}: & \mathsf{udp\_read}' & \mapsto \\
 & \multicolumn{2}{l}{\quad (\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}, \mathrm{cookie} : \mathsf{cookie}\} \rightarrow \mathsf{int}, \lambda(x)\ldots.)} \\
 & \mathsf{udp\_write}' & \mapsto \\
 & \multicolumn{2}{l}{\quad (\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}, \mathrm{cookie} : \mathsf{cookie}\} \rightarrow \mathsf{int}, \lambda(x)\ldots.)} \\
 & \mathsf{sockh\_coer} & \mapsto \quad ((\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}\} \rightarrow \mathsf{int}) \rightarrow \\
 & \multicolumn{2}{l}{\quad (\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}, \mathrm{cookie} : \mathsf{cookie}\} \rightarrow \mathsf{int}),} \\
 & \multicolumn{2}{l}{\quad \lambda(f).\mathbf{if}\ \mathrm{f} = \mathsf{udp\_read}\ \mathbf{then}\ \mathsf{udp\_read}'\ \mathbf{else\ if}\ \mathrm{f} = \mathsf{udp\_write}\ \mathbf{then}\ \mathsf{udp\_write}')} \\
 & \mathsf{sock\_coer} & \mapsto \quad \ldots \\
 & \mathsf{security\_info} & \mapsto \quad (\mathsf{int} \rightarrow \mathsf{cookie}, \lambda(x)\ldots)
\end{array}
$$

Figure 5.11: A sample update to the I/O kernel

that contain handler objects (such as the table used by getsockhandler) will be updated to refer to the new versions. If any code in the program called udp_read or udp_write directly, we could replace them with *stub* functions [Hic01, FS91], forwarding calls to the new version, and filling in the added argument. Thus, Proteus indirectly supports updating functions to new types for those rare occasions when this is necessary.

5.4 Operational Semantics

The operational semantics is defined using rewriting rules between *configurations*, which are triples consisting of a type environment Ω, a heap H and an expression e, as shown in Figure 5.12.

The type environment Ω defines a configuration's named types. Each type in $\mathrm{dom}(\Omega)$ maps to a single representation τ; some related approaches [Dug01, Hic01] would permit t to map to a set of representations indexed by a version. We refer to our non-versioned approach as being *representation consistent* since a running program has but one definition of a type at any given time.

The heap H is a map from heap addresses ρ to pairs (ω, b), where ω is a *type tag* and b is a binding. We use the heap to store both mutable references created with **ref** and top-level bindings created with **var** and **fun**; therefore ρ ranges over locations r and external names z. For locations, the type tag ω is simply $\cdot$, indicating the absence of a type, and for identifiers, e.g. z, it is the type τ which appeared in the definition

Syntax

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

Updates

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

Figure 5.12: Syntax for dynamic semantics

of z in the program. Type tags are used to type check new and replacement definitions provided by a dynamic update. As there is no type tag associated with references, there is no runtime overhead to them. They merely act as an interface for updates.

Configuration evaluation is defined by the judgement $\Omega; H; e \longrightarrow \Omega; H'; e'$, shown in Figure 5.13. This judgement consists of a series of computations, the order of which is determined by evaluation contexts. All expressions e can be uniquely decomposed into $\mathbb{E}[e']$ for some evaluation context $\mathbb{E}$ and e', so the choice of computation rule is unambiguous. A program P is compiled into a configuration $\Omega; H; e = \mathcal{C}(\emptyset; \emptyset; P)$, as shown at the bottom of Figure 5.13.

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

$\boxed{\textbf{Computation: } H; e \longrightarrow H'; e'}$

$H; \{l_1 = v_1, \ldots, l_n = v_n\}.l_i \longrightarrow H; v_i$ (PROJ)

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

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

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

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

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

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

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

$\boxed{\textbf{Configuration Evaluation: } \Omega; H; e \xrightarrow{\cdot/\text{upd}} \Omega'; H'; e'}$

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

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

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

UPDATE is not subject to closure under CONG.

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

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

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

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

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

Figure 5.13: Dynamic semantics for Proteus$^{\mathbf{con}}$

$\boxed{\textbf{Dynamic Updating: } \mathcal{U}[-]^{\text{upd}}}$

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

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

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

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

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

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

Figure 5.14: $\mathcal{U}[-]^{-}$ definition for heaps and expressions

5.4.1 Replacing Top-level Identifiers

A top-level identifier z from the source program is essentially a statically-allocated reference cell. As a result, at update-time we can change z's binding in the heap; afterwards any code that accesses (dereferences) z will see the new version. However, our treatment of references differs somewhat from the standard one to facilitate dynamic updates.

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

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

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

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

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

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

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

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

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

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

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

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

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

The case for named type lists ($\hat{\Omega}$) is the same

Figure 5.15: $\mathcal{U}[-]^{-}$ definition for contexts

5.4.2 Updating Data of Named Type

As discussed in §5.1, $\mathrm{Proteus}^{\mathbf{con}}$ uses coercions to identify where data of a type t is being used abstractly and concretely. The (CONABS) rule allows an abstract value $\mathbf{abs}_{\mathsf{t}}\ v$ to be used concretely when it is provided to $\mathbf{con}_{\mathsf{t}}$; this annihilates both coercions so that v can be used directly.

At update time, given a type transformation function c for an updated type t, we rewrite each occurrence $\mathbf{abs}_{\mathsf{t}}\ e$ to be $\mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ e)$. Although only values can be stored in the heap initially, heap values of the form $\mathbf{abs}_{\mathsf{t}}\ v$ will be rewritten to be $\mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ v)$, which is no longer a value. Therefore, $!r$ can potentially dereference an expression from the heap. While this is not a problem in itself, the transformation should be performed only once since it conceptually modifies the data in place. Therefore, the (DEREF) rule evaluates the contents of the reference and then *writes back* the result before proceeding. Whether the coercions be in the heap or the program, when they are executed is (for all intents are purposes) unpredictable. As a result, coercions should be written to act locally and avoid side-effecting computation. One could imagine enforcing this, but we do not do so here.

5.4.3 Update Semantics

A dynamic update upd is modelled with a *labelled transition*, where upd labels the arrow. When no update is available, an $\mathbf{update}$ expression simply evaluates to 1, by (NO-UPDATE). Otherwise, (UPDATE) specifies that if upd is well-formed (by $\mathrm{updateOK}(-)$), the $\mathbf{update}$ evaluates to 0, and the program is updated by transforming the current type environment, heap, and expression according to $\mathcal{U}[-]^{\mathrm{upd}}$. This transformation is defined in Figures 5.14 and 5.15. When transforming expressions, $\mathcal{U}[-]^{\mathrm{upd}}$ applies type transformation functions to all $\mathbf{abs}_{\mathsf{t}}\ e$ expressions of a named type t that is being updated. When transforming the heap, it replaces top-level identifier definitions with their new versions, and adds all of the new bindings. When transforming Ω (Figure 5.15), it replaces type definitions with their new versions, and inserts new definitions into specified slots in the list.

5.5 Update Safety

The conditions placed upon an update to guarantee type-safety are formally expressed by the precondition to the (UPDATE) rule given in Figure 5.16. The $\mathrm{updateOK}(-)$ predicate must determine that it is valid to apply the update at the current point —a dynamic

$\boxed{\text{updateOK}(\text{upd}, \Omega, H, e)}$

$$
\begin{aligned}
&\text{updateOK}(\text{upd}, \Omega, H, e) = \\
&\quad \text{conFree}[\,H\,]^{\text{upd}} \wedge \\
&\quad \text{conFree}[\,e\,]^{\text{upd}} \wedge \\
&\quad \Gamma = \text{types}(H) \wedge \\
&\quad \vdash \mathcal{U}[\Omega]^{\text{upd}} \wedge \\
&\quad \forall \mathsf{t} \mapsto (\tau, \mathsf{c}) \in \text{upd.UN}. \quad \mathcal{U}[\Omega, \Gamma]^{\text{upd}} \vdash \mathsf{c} : \Omega(\mathsf{t}) \to \tau \wedge \\
&\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \text{upd.UB}. \quad \mathcal{U}[\Omega, \Gamma]^{\text{upd}} \vdash b_v : \tau \wedge \\
&\qquad\quad \text{heapType}(\tau, b_v) = \Gamma(\mathsf{z}) \wedge \\
&\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \text{upd.AB}. \quad \mathcal{U}[\Omega, \Gamma]^{\text{upd}} \vdash b_v : \tau \wedge \mathsf{z} \notin \text{dom}(H)
\end{aligned}
$$

$\boxed{\text{conFree}[\,-\,]^{\text{upd}} = \mathbf{tt} \mid \mathbf{ff}}$

$$
\begin{aligned}
&\text{conFree}[\,\mathsf{z} = (\tau, b), H\,]^{\text{upd}} \\
&\qquad = \text{conFree}[\,H\,]^{\text{upd}} \wedge \begin{cases} \mathbf{tt} & \text{if } \mathsf{z} \in \text{dom}(\text{upd.UB}) \\ \text{conFree}[\,b\,]^{\text{upd}} & \text{otherwise} \end{cases} \\
&\text{conFree}[\,r = (\cdot, e), H\,]^{\text{upd}} = \text{conFree}[\,e\,]^{\text{upd}} \wedge \text{conFree}[\,H\,]^{\text{upd}}
\end{aligned}
$$

$$\text{conFree}[\,n\,]^{\text{upd}} = \mathbf{tt} \qquad \text{conFree}[\,x\,]^{\text{upd}} = \mathbf{tt}$$

$$\text{conFree}[\,\mathbf{con}_{\mathsf{t}}\ e\,]^{\text{upd}} = \begin{cases} \mathbf{ff} & \text{if } \mathsf{t} \in \text{dom}(\text{upd.UN}) \\ \mathbf{tt} & \text{otherwise} \end{cases}$$

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

$\boxed{\text{types}(H) = \Phi}$

$$
\begin{aligned}
&\text{types}(\emptyset) &&= \emptyset \\
&\text{types}(\mathsf{z} \mapsto (\tau \to \tau', \lambda(x).e\ ), H') &&= \mathsf{z} : \tau \to \tau', \text{types}(H') \\
&\text{types}(\mathsf{z} \mapsto (\tau, e), H') &&= \mathsf{z} : \tau\ \mathbf{ref}, \text{types}(H')
\end{aligned}
$$

$\boxed{\text{heapType}(\tau, b_v) = \mathbf{tt} \mid \mathbf{ff}}$

$$
\begin{aligned}
&\text{heapType}(\tau_1 \to \tau_2) &&= \tau_1 \to \tau_2 \\
&\text{heapType}(\tau) &&= \tau\ \mathbf{ref} \qquad \text{where } \tau \neq \tau_1 \to \tau_2
\end{aligned}
$$

Figure 5.16: $\text{updateOK}(-)$, $\text{conFree}[\,-\,]^{-}$, $\text{types}(-)$ and $\text{heapType}(\tau, b_v)$ definitions

property—and that the update is compatible with the program. The latter is a static property, in the sense that the information to perform it is available without recourse to the current state of the program, provided one has the original source and the updates previously applied.

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

Figure 5.17: Example active expression

5.5.1 Update Timing

To clarify the importance of timing, Figure 5.17 shows the expression fragment of our example program after some evaluation steps (the outer **let** i = ... binding comes from loop and the argument to post is the partially-evaluated dispatch function). The **let** u2 = **update**... is in redex position, and suppose that the update described in §5.3 is available, which updates sockhandler to have an additional cookie argument, amongst other things. If this update were applied, the user's type transformer sockh_coer would be inserted to convert udp_read and would be called next. Evaluating the transformer replaces udp_read with udp_read′, and applying (CONABS) yields the expression udp_read′$(v_{\mathsf{sock}}, (\mathbf{con}_{\mathsf{req}}\ v_{\mathsf{req}}).\mathrm{buf}, v_{\mathsf{sflags}})$. But this would be type-incorrect! The new version udp_read′ expects a fourth argument, but the existing call only passes three arguments.

The problem is that at the time of the update the program is evaluating the old version of dispatch, which expects sockhandler values to take only three arguments. That is, this point in the program is not "con-t-free" since it will manipulate t values concretely. This fact is made manifest by the usage of $\mathbf{con}_{\mathsf{sockhandler}}$ in the active expression. In general, we say a configuration $\Omega; H; e$ is *con-free* for an update upd if for all named types t that the update will change, $\mathbf{con}_{\mathsf{t}}$ is not a subexpression of the active expression e or any of the bindings in the heap that are not replaced by the update. We write this as $\mathrm{conFree}[-]^{\mathrm{upd}}$; the definition is given in Figure 5.16.

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

5.5.2 Update well-formedness

The conditions for update well-formedness are part of the $\mathrm{updateOK}(-)$ predicate, defined in Figure 5.16. In addition to checking proper timing with the $\mathrm{conFree}[-]$ checks, this predicate ensures that type-safety is maintained following the addition or replacement of code and types. The $\mathrm{types}(H)$ predicate extracts all of the type tags from H and constructs a suitable Γ for typechecking the new or replacement bindings. Since heap objects are stored with their declared type τ, if they are non-functions then in Γ they are given type τ **ref**. Next, the updated type environment $\mathcal{U}[\Omega]^{\mathrm{upd}}$ is checked for well-formedness. Then, using the updated Ω and Γ, we check that the type transformer functions, replacement bindings and new bindings are all well-typed. These type-checks apply only to expressions contained in the update—none of the existing code must be rechecked (though its types, as stored in the heap, are needed to check the new code).

5.6 Properties

$\mathrm{Proteus}^{\mathbf{con}}$ enjoys an essentially standard type safety result. To state it we need a notion of configuration typing. This is expressed by the judgements $\vdash \Omega; H; e : \tau$ and $\Omega; \Phi \vdash H$, defined in Figures 5.19 and 5.18 respectively. Configuration well-formedness is predicated on the existence of some Φ, called the *heap interface*, that properly maps external names z and references r to types τ. That is, a configuration is well-formed as long as there exists some Φ sufficient to type check the heap ($\Omega; \Phi \vdash H$) and to type check the active expression. Note that we write Ω, Φ to denote the concatenation of the heap interface and the configuration type environment, which defines the Γ used to type check the active expression e.

The definition of heap typing is shown at the bottom of Figure 5.19. It establishes two facts: (1) each of the types in Φ accurately represents the types of the bindings found in H; (2) each of the bindings in the heap type checks under Φ and the current type environment Ω. For functions, we type check with the updateability indicated by the function's type, while for other bindings we assume N. Note that the approach of assuming the existence of a Φ, is necessary to allow cycles in the reference graph.

$\boxed{\textbf{Type list: } \Omega \vdash \Omega}$

Note: we overload notation. When Ω is on the lhs of $\vdash$ it is a finite partial function and on the rhs it is a list

$$\Omega \vdash \emptyset \qquad \text{(A.TYPE.TENV.EMPTY)}$$

$$\frac{\Omega, \mathsf{t} = \tau \vdash \Omega' \qquad \Omega \vdash \tau}{\Omega \vdash \mathsf{t} = \tau, \Omega'} \qquad \text{(A.TYPE.TENV.DEF)}$$

Figure 5.18: Type environment well-formedness definition

$\boxed{\textbf{Configuration typing: } \Gamma \vdash \Omega; H, e : \tau}$

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

$\boxed{\textbf{Heap typing: } \Omega; \Phi \vdash H}$

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

Figure 5.19: Configuration and heap typing

The type environment Ω must be consistent. This is particularly important when an update is applied as we must ensure that the resulting type environment is valid. The rules in Figure 6.2 ensure this by requiring all types mentioned in other types to be both defined and linearly orderable (non-recursive). However, it is not hard to add recursive types to our system (see Section 6.6.2)

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

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

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

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

Proof sketches appear in Appendix A.

5.7 Conclusion

In this chapter we have presented Proteus, a simple calculus for modelling type-safe dynamic updates in C-like languages. To ensure that updates are type-safe in the presence of changes to named types, Proteus exploits the idea of "con-t-freeness": a given update point is con-t-free if the program will never use a value of type t concretely at its old representation from then on. The solution we presented is based on explicit coercions from named types to their representations. We gave a fully automatic and deterministic way to insert such coercions based on coercive subtyping and showed that the resulting notion of con-freeness can be checked dynamically.

6
Update Capability Analysis

Type safety for the system described in Chapter 5 is predicated on a dynamic con-free check. Unfortunately, the unpredictability of this dynamic con-free check could make it hard to tell whether an update failure is transient (meaning the update is not valid in this program state) or permanent (meaning the update is invalid in all program states), since the dynamic check is for a particular program state. Rather, we would prefer to reason about update behaviour *statically*, to (among other things) assess whether there are enough update points.

This chapter introduces a way to statically determine which types are updatable at each update point as an analysis on $\mathrm{Proteus}^{\mathbf{con}}$ programs. For each $\mathbf{update}$ expression, we estimate those types t for which the program may *not* be con-t-free. We annotate the $\mathbf{update}$ with those types, and at run time ensure that any dynamic update at that point does not change them. This is both simpler than the con-free dynamic check and more predictable. In particular, we can automatically infer those points at which the program is con-free for all types t, precluding dynamic failure. In other words, we eliminate the need for $\mathrm{conFree}[-]$ and we make the update behaviour of the program easier to reason about, since many acceptable update points are known statically. We present the *updateability analysis* as a type and inference system, and establish its soundness.

6.1 Capabilities

Our goal is to define and enforce a notion of con-freeness for a *program*, rather than a *program state*. In other words, we wish to determine for a particular $\mathbf{update}$ whether

$$
\begin{array}{rlll}
\text{Capabilities} & \Delta & ::= & \{\mathsf{t}_1, \ldots \mathsf{t}_\mathsf{n}\} \mid \Delta \cap \Delta \\
\text{Updateability} & \mu & ::= & \mathsf{U} \mid \mathsf{N} \\
\text{Types} & \tau & ::= & \cdots \mid \tau \xrightarrow{\mu;\Delta} \tau \\
\text{Expressions} & e & ::= & \cdots \mid \mathbf{update}^{\Delta} \\
\text{Programs} & P & ::= & \cdots \mid \mathbf{fun}\ \mathsf{z_1}^{\mu_1;\Delta_1;\Delta'_1}(x : \tau_1) : \tau'_1 = e_1\ \mathbf{and}\ \ldots \\
 & & & \quad \mathbf{fun}\ \mathsf{z_n}^{\mu_n;\Delta_n;\Delta'_n}(x : \tau_n) : \tau'_n = e_n\ \mathbf{in}\ P \\
\text{Heap expressions} & b & ::= & e \mid \lambda^{\Delta}(x).e \\
\text{Heap values} & b_v & ::= & v \mid \lambda^{\Delta}(x).e
\end{array}
$$

Figure 6.1: Extended syntax for $\mathrm{Proteus}^{\Delta}$

it will be acceptable to update some type t. An update to t will be unacceptable if an occurrence of $\mathbf{con}_\mathsf{t}$ exists in any old code evaluated in the continuation of the $\mathbf{update}$. Assuming we can discover all such occurrences of $\mathbf{con}_\mathsf{t}$, we annotate $\mathbf{update}$ with those types t, indicating that they should not be updated. To determine this we associate a pre and post *capability* with every syntactic constructor of the language. A capability is a set of named types denoted Δ with the informal meaning that whenever t is in the capability then values of type t can be concreted. The annotation on $\mathbf{update}$ points has a negative effect on the post capability associated with it since it serves as a bound on what types may be used concretely in the continuation of the $\mathbf{update}$. That is, any code following an $\mathbf{update}$ must type check with the capability of the annotation on $\mathbf{update}$. Since an $\mathbf{update}$ could change only types not in the capability, we are certain that existing code will remain type-safe. As a consequence, if we can type-check our program containing only $\mathbf{update}$ points with empty annotations, we can be sure that no update will fail due to bad timing, i.e. if an update succeeds at one update point then it will succeed at all.

6.2 Typing

We define a capability type system that tracks the capability at each program point to ensure that $\mathbf{update}$s are annotated soundly. To do this, we introduce a new language, $\mathrm{Proteus}^{\Delta}$, that differs from Proteus only in that types, functions, and $\mathbf{update}$ are annotated with capabilities. The syntax changes are given in Figure 6.1. We must also adjust compilation (defined in Figure 5.13) in the case of functions to add the necessary

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

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

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

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

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

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

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

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

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

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

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

Figure 6.2: Expression judgements for updateability analysis (part I)

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

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

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

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

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

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

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

Figure 6.3: Expression judgements for updateability analysis (part II)

annotation on the generated binding and type:

$$\mathcal{C}\left(\Omega; H; \begin{array}{l}\textbf{fun}\ \mathsf{f_1}^{\mu_1;\Delta_1;\Delta'_1}(x : \tau_1) : \tau'_1 = e_1\ \textbf{and} \\ \textbf{fun}\ \mathsf{f_n}^{\mu_n;\Delta_n;\Delta'_n}(x : \tau_n) : \tau'_n = e_n\ \textbf{in}\ P\end{array}\right) =$$
$$\mathcal{C}(\Omega; H, \mathsf{f_1} \mapsto (\tau_! \xrightarrow{\mu_1;\Delta'_1} \tau'_1, \lambda^{\Delta_1}(x).e_1\ , \ldots \mathsf{f_n} \mapsto (\tau_n \xrightarrow{\mu_n;\Delta'_n} \tau'_n, \lambda^{\Delta_n}(x).e_n\ ); P)$$

For the remainder of this section, we consider the type system for $\mathrm{Proteus}^{\Delta}$, covering judgements for expressions, programs, and configurations.

6.2.1 Expression Typing

The rules for typing expressions are given in Figures 6.2 and 6.3, defining judgement $\Delta; \Gamma \vdash_{\mu} e : \tau; \Delta'$. This can be read as with capability Δ in the environment Γ and updateability μ expression e has type τ and results in capability Δ'. We call Δ the *precapability* and Δ' the *postcapability*. The *updateability* μ that parametrises each rule indicates whether that expression *may* cause a dynamic update, U indicating updates may occur and N indicating that they may not. Updateabilities are used to rule out dynamic updates in undesirable contexts, as we explain in the next subsection.

Typing $\mathbf{update}$ and $\mathbf{con}_{\mathsf{t}}\ e$

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

To type check dispatch in Figure 5.3, we must annotate the $\mathbf{update}$ in $\mathbf{let}\ \mathrm{u1} = \mathbf{update}\ \mathbf{in}\ \ldots$ with a capability $\{\mathsf{fdtype}, \mathsf{req}, \mathsf{sockhandler}\}$, since these types are used by $\mathbf{con}$ expressions following that point within dispatch. By the same reasoning, the annotation on the u2 update would be $\{\mathsf{req}, \mathsf{sockhandler}\}$, and the u3 update annotation can be empty. The (A.EXPR.UPDATE) rule requires updateability U; updates cannot be performed in a non-updatable (N) context.

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

Function calls

Function types have an annotation $\mu; \hat{\Delta}$, where $\hat{\Delta}$ is the *output capability*. If calling a function could result in an update, the updateability μ must be U. Returning to our motivating example in Figure 5.3 and using the annotations on **update** mentioned above the type for dispatch would be req $\xrightarrow{\mathsf{U};\emptyset}$ handResult. In the (A.EXPR.UPDATE) rule, the output capability is bounded by the annotation on the **update**; in the (A.EXPR.APP) rule, if the callee can perform an update then the caller's output capability Δ''' is bounded by the callee's output capability $\hat{\Delta}'$ for the same reason. This is expressed in the conditional constraint $\hat{\mu} = \mathsf{U} \implies \Delta''' \subseteq \hat{\Delta}$. The updateability constraint $\hat{\mu} \leq \mu$ can be read as, if the callee cannot perform an update ($\mu = \mathsf{N}$) then it cannot call a function that causes one ($\hat{\mu} \neq \mathsf{U}$). Whenever the called function cannot perform an update, the rule places no restriction on the caller's capability or updateability. We will take advantage of this fact in how we define type transformer functions, described below.

A perhaps interesting effect of (A.EXPR.APP) is that a function f's output capability must mention those types used concretely by its callers following calls to f. To illustrate, say we modify the type of post in Figure 5.2 to be int $\rightarrow$ int rather than handResult $\rightarrow$ int. As a result, loop would have to concrete the handResult returned by dispatch before passing it to post, resulting in the code

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

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

Another unintuitive aspect of (A.EXPR.APP) is that to call a function, we would expect that the caller's capability must be compatible with (i.e., must be a superset of) the function's input capability, but this condition is not necessary. Instead, the type system assumes that all calls will be to a function's most recent version, which will be guaranteed at update-time to be compatible with the program's type definitions (see §6.3). In effect, the type system approximates, for a given update point, the concretions in code that an updating function could *return to*, but not code it will later call, which is guaranteed to be safe. This is critical to avoid restricting updates unnecessarily.

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

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

$$\frac{\begin{array}{l}\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma \vdash \tau_1' <: \tau_2' \\ \Delta_2 \subseteq \Delta_1 \qquad \mu_1 \leq \mu_2\end{array}}{\Gamma \vdash \tau_1 \stackrel{\mu_1;\Delta_1}{\longrightarrow} \tau_1' <: \tau_2 \stackrel{\mu_2;\Delta_2}{\longrightarrow} \tau_2'} \qquad \text{(A.SUB.FUN)}$$

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

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

Figure 6.4: Subtyping judgement for updatability analysis

Other Rules

Unlike $\mathbf{con}_{\mathrm{t}}\ e$ expressions, $\mathbf{abs}_{\mathrm{t}}\ e$ expressions place no constraint on the capability (see rule (A.EXPR.ABS)). This is because a dynamic update that changes the definition of t from τ to τ' requires a well-typed type transformer c to rewrite $\mathbf{abs}_{\mathrm{t}}\ e$ to $\mathbf{abs}_{\mathrm{t}}\ (\mathsf{c}(e))$, which will always be well-typed assuming suitable restrictions on c to be described in section 6.3.

Subtyping

The type system permits subtyping via the (A.EXPR.SUB) rule, which also permits coarsening (making smaller) of the output capability Δ. Intuitively, this coarsening is always sound because it will put a stronger restriction on limits imposed by prior updates. The subtyping rules shown in Figure 6.4 adds flexibility to programs and to their updates. The interesting rule is (A.SUB.FUN) for function types. Output capabilities are contravariant: if a caller expects a function's output capability to be Δ, it will be a conservative approximation if the function's output capability is actually larger. A function that performs no updates can be used in place of one that does, assuming they have compatible capabilities.

Why a Capability Type System and Not an Effect System?

A type system defines a relation $\vdash$ that in its simplest form is a tertiary relation between contexts, expressions and types. An effect system makes this a quaternary relation between contexts, expressions, types and effects. A capability type system makes this a quintuple relation between contexts, expressions, types, pre-capabilities and post-capabilities. As tuples of a given size are isomorphic up to permutations in the order, an effect system with two effect sets is isomorphic to a capability type system. The point being that, semantically, it does not matter whether we write $\Delta; \Gamma \vdash e : \tau; \Delta'$ or $\Gamma \vdash e : \tau; \Delta; \Delta'$. Thus the question of why do we need a capability type system instead of an effect system is reduced to why we need two effects.

The objective of our type system is to check whether prohibiting the change of a given set of types at a particular update point is sufficient to ensure safety. To do this we must know the types that might be updated, as these types must not be concreted by subsequent execution that assumes the pre-update representation. In order to ensure this latter condition we must know the types used concretely by an expression. Thus, we need to know two pieces of information, the types updated by an expression and those concreted by it. Hence two effect sets.

Instead of thinking in terms of effects, we choose two think in terms of capabilities, believing it more intuitive. As we have seen so far in this chapter, we still associate two sets with an expression, but they stand for the types that can be concreted (Δ) and those that the expression does not update (Δ'). The set Δ tells us something about the environment, so we write it on the left of the turnstyle, as these usually denote properties of the environment. The set Δ' tells us something about the expression under consideration and so we write it on the right, as objects on the right usually tell us something about the expression.

6.2.2 Program Typing

The rules for typing programs are given in Figure 6.5, defining the judgement $\Gamma \vdash_P P : \tau$. The (A.PROG.TYPE) rule adds a new type definition to the global environment, and the (A.PROG.FUN) rule simply checks the function's body using the capabilities and updateability defined by its type. Since v is a value and cannot effect an update, the (A.PROG.VAR) rule checks it with an empty capability Δ and updateability N. Finally, the (A.PROG.EXPR) rule type checks the body of the program using an arbitrary capability and updateability U to allow updates.

$\boxed{\textbf{Program Typing: } \Gamma \vdash_P P : \tau}$

$$\frac{\begin{array}{l}\Gamma, \mathsf{t} = \tau' \vdash_P P : \tau \\ \Gamma \vdash \tau'\end{array}}{\Gamma \vdash_P \textbf{type}\ \mathsf{t} = \tau'\ \textbf{in}\ P : \tau} \quad \text{(A.PROG.TYPE)}$$

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

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

$$\frac{\Delta; \Gamma \vdash_{\mathsf{U}} e : \tau; \Delta'}{\Gamma \vdash_P e : \tau} \quad \text{(A.PROG.EXPR)}$$

Figure 6.5: Program judgements for updatability analysis

6.2.3 Configuration Typing

To prove soundness syntactically (Section B), we need typing judgements to express the well-formedness of configurations. The well-formedness judgement is that of Figure 5.18 from the previous chapter. Configuration typing is altered to take account of capabilities but otherwise remains unchanged and is given in Figure 6.6.

6.3 Operational Semantics

The dynamic semantics from Figure 5.13 remains unchanged with the exception of the $\mathrm{updateOK}(-)$ predicate for (UPDATE), shown in Figure 6.7. The two timing-related changes are highlighted by the boxes labelled (a) and (b). First, Δ, taken from $\textbf{update}^{\Delta}$, replaces e as the last argument. This is used in (a) to syntactically check that no types mentioned in Δ are changed by the update. Change (a) also refers to $\mathrm{bindOK}[\,\Gamma\,]^{\mathrm{upd}}$

$\boxed{\textbf{Configuration typing: } \Gamma \vdash \Omega; H, e : \tau}$

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

$\boxed{\textbf{Heap typing: } \Omega; \Phi \vdash H}$

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

Figure 6.6: Configuration typing

to ensure that all top-level bindings in the heap that use types in upd.UN concretely, as indicated by their input capability, are also replaced. This allows the type system to assume that calling a function is always safe, and need not impact its capability. Together, these two checks are analogous to the con-free dynamic check to ensure proper timing.[1]

Type transformers provided for updated types must not, when inserted, violate assumptions made by the updateability analysis. In particular, each $\mathbf{abs}_{\mathsf{t}}\ e$ appearing in the program type checks with some capability prior to an update, i.e., $\Delta; \Gamma \vdash_{\mu} \mathbf{abs}_{\mathsf{t}}\ e : \tau; \Delta'$. If type t is updated with transformer c, we require $\Delta; \Gamma \vdash_{\mu} \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ e) : \tau; \Delta'$. Since $\mathbf{abs}_{\mathsf{t}}\ e$ expressions could be anywhere at update time, and could require a different capability Δ to type check, condition (b) conservatively mandates that transformers c must check in an empty capability, and may not perform updates (c's type must have updateability N). These conditions are sufficient to ensure type correctness. Otherwise, a transformer function c is like any other function. For example, if it uses some type t concretely, it will have to be updated if t is updated. The ramifications of this fact are explored in §6.6.

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

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

$$\boxed{\mathrm{bindOK}[\, H \,]^{\mathrm{upd}} = \mathbf{tt} \mid \mathbf{ff}}$$

$$\mathrm{bindOK}[\, \emptyset \,]^{\mathrm{upd}} = \mathbf{tt}$$

$$
\begin{array}{l}
\mathrm{bindOK}\left[\, \mathsf{z} \mapsto (\tau_1 \overset{\mu;\Delta'}{\longrightarrow} \tau_2, \lambda^{\Delta}(x).e\ ), H' \,\right]^{\mathrm{upd}} = \mathrm{bindOK}[\, H' \,]^{\mathrm{upd}} \wedge \\
\quad (\mathrm{dom}(\mathrm{upd.UN}) \cap \Delta \neq \emptyset) \implies \mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})
\end{array}
$$

$$\mathrm{bindOK}[\, \mathsf{z} \mapsto (\tau, b), H' \,]^{\mathrm{upd}} = \mathrm{bindOK}[\, H' \,]^{\mathrm{upd}} \wedge \tau \neq \tau_1 \overset{\mu;\Delta'}{\longrightarrow} \tau_2$$

$$\mathrm{bindOK}[\, r \mapsto (\cdot, b), H' \,]^{\mathrm{upd}} = \mathrm{bindOK}[\, H' \,]^{\mathrm{upd}}$$

Figure 6.7: Precondition for $\mathbf{update}^{\Delta}$ operational rule

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

```
fun f($x$ : t) : t′ =
  let $y$ = con$_{t''}$ (abs$_{t''}$ 1) in
  let $z$ = con$_{t}$ $x$ in abs$_{t'}$ $(z + y)$
```

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

6.3.1 Phase Order

The key advantages of $\mathrm{Proteus}^{\Delta}$ over Proteus are two-fold. First, the updatability of a program can be assessed before the program is run, avoiding the possibility of writing a program only to discover later, at runtime, that no updates are valid. The second advantage over Proteus programs comes from a change in the phase order. In proteus, all of the updateOK predicate has to be checked at runtime. However, in $\mathrm{Proteus}^{\Delta}$ all but the bindOK clause of updateOK can be checked off-line. In order to do this we have only to assume knowledge of the typing of the original program and the particular update point at which the update will be applied. In practice, the ability to perform the majority of checks off-line will significantly ease development of updatable programs.

6.4 Inference

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

As a result of these changes, conditions imposed on capability variables by the typing and subtyping rules become simple set and term constraints [Hei92]. We begin with a typing derivation and extract any constraints generated by the expression or subtyping rules used. For example, say we use the function subtyping rule in the derivation to deduce

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

then we introduce the constraints

$$\begin{array}{l}\Gamma \vdash \tau_2 <: \tau_1 \\ \Gamma \vdash \tau_1' <: \tau_2' \\ \varphi_2 \subseteq \varphi_1 \\ \varepsilon_1 \leq \varepsilon_2\end{array}$$

where the subtype constraints expand themselves to more basic constraints. All of the final constraints are of one of the following forms.

Updatability constraints $\varepsilon = \mathsf{U}$ and $\mu \leq \mu'$

Implicational constraints $\mu = \mathsf{U} \implies C$

Capability constraints $\varphi \subseteq \Delta$ and $\mathrm{t} \in \Delta$.

A solution consists of a substitution σ, which is a map from variables φ to capabilities $\{\mathrm{t}_1, \ldots, \mathrm{t}_n\}$, and from variables ε to updateabilities either U or N. The constraints can be solved efficiently with standard techniques in time $O(n^3)$ in the worst case (but far better on average), where n is the number of variables φ or set constants $\{\cdot\}$ mentioned in the constraints.

For updateabilities, we want the *greatest* solution; that is, we want to allow as many functions as possible to perform updates (with an unannotated program, this will vacuously be the case). For the capabilities, we are interested in the *least* solution, in which we minimise the set to substitute for φ, since it will permit more dynamic updates. For $\mathbf{update}^{\varphi}$, a minimal φ imposes fewer restrictions on the types that may be updated at that point. For functions $\tau \xrightarrow{\varepsilon;\varphi'} \tau'$, the smaller φ' imposes fewer constraints on subtypes, which in turn permits more possible function replacements.

Algorithmically, constraint solving proceeds as follows. First, we find the greatest solution for the updatability constraints. This gives the greatest flexibility when applying future updates as functions that perform update can be replaced with those that cannot; the converse is not true. Using this solution we resolve the implicational constraints $\mu = \mathsf{U} \implies C$, adding C to the capability constraint set whenever $\mu = \mathsf{U}$. The capability constraints are then solved for the least solution.

When using inference for later versions of a program, we must introduce subtyping constraints between an old definition's (solved) type and the new version's to-be-inferred one. This ensures that the new definition will be a suitable dynamic replacement for the old one.

6.4.1 Inferring update Points

Using the inference system, we can take a program that is devoid of $\mathbf{update}$ expressions, and infer places to insert them that are con-free for all types. Define a source-to-source rewriting function rewrite : $P \to P'$ that inserts $\mathbf{update}^{\varphi}$ at various locations throughout the program for example, before every function return point or after every function call. Then we perform inference, and remove all occurrences of $\mathbf{update}^{\varphi}$ for which φ is not $\emptyset$ (call these *universal* update points as they do not restrict the types that may be updated). Adding more points implies greater availability, but longer analysis times and more runtime overhead. The inference is stable under adding more update points because the annotations φ on them are unaffected by those on other update points; rather they are only influenced by occurrences of $\mathbf{con}$ in their continuations.

6.5 Properties

We have the usual type soundness property for the calculus.

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

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

(ii) e *is a value*

The proof of this Theorem is shown in Appendix B. The proof is based on standard techniques for proving syntactic type soundness [WF94], but extended to deal with a capability type system. Proving soundness for the system guarantees that our inferred update points are safe. Consequently, any update judged suitable by $\mathrm{updateOK}()$ at runtime will not invalidate the type of the program.

The idea of update capability weakening (Lemma B.1.13), plays a key rôle in the proof. Update capability weakening shows that the Δ annotation on updates is faithful to our intended meaning, that is, given an update point $\mathbf{update}^{\hat{\Delta}}$ in redex position in some larger expression, the only types used concretely following the update are contained in $\hat{\Delta}$. More formally, if

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

then

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

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

6.6 Extensions

This section considers two extensions to our basic approach: binding deletion and recursive types.

6.6.1 Binding Deletion

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

1. Dead bindings will unnecessarily consume virtual memory, which could be problematic over time.

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

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

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

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

6.6.2 Recursive types

The source language we presented uses simple user-defined types. In particular, these types are non-recursive. However, iso-recursive types are trivial to add to our system as $\mathbf{con}_{\mathrm{t}}\ e$ and $\mathbf{abs}_{\mathrm{t}}\ e$ correspond precisely to the mediating coercions $\mathbf{fold}_{t}\ e$ and $\mathbf{unfold}_{t}\ e$ of iso-recursive types.

6.7 Conclusions

In this chapter we defined $\mathrm{Proteus}^{\Delta}$, the first system to give static assurances over which types will be updatable at runtime. The analysis to determine this property was presented as a novel capability-based type system, for which we showed there was a decision procedure. We gave an operational semantics and proved type soundness, thus ensuring the representation consistency of updates.

7

Implementation

In previous chapters we built up an underlying theory for languages that support dynamic update. In this chapter we apply the Proteus theory to the design and implementation of a whole program, source-to-source transform for applications written in the C programming language. The main technical problems lie in how to deal with the unsafe features of C and in designing an efficient runtime to support dynamic update. We show that our prototype implementation scales, transforming substantial C programs such as VSFTPD[1], a secure FTP server; Apache[2], a popular web server; and OpenSSH[3], a secure remote shell. We deal with the full C language, making conservative assumptions for features of C that are unsafe.

This chapter begins by looking at how to write and update updatable programs. A general overview of the architecture is given and the update process is described at the user level. Subsequently, specific details for each stage of the transformation are given highlighting the problems specific to C. Having described how to compile an initial updatable program we look at how patch compilation differs. Finally we give performance results and conclude.

7.1 Writing Updatable Programs

In this section we explain how the Proteus notion of update can be used in practice to achieve our goal of writing and evolving computer programs without the need for

[1]`http://vsftpd.beasts.org`
[2]`http://www.apache.org`
[3]`http://www.openssh.com`

down-time. Our approach can be summarised as write-run-patch. That is, write an initial program, run that program, then write and apply dynamic patches. Our patches are notionally similar to classical program patches that are commonly applied to source trees using `patch`[4], in that they embody the change set from one version of a program to another. Of course, as we patch a running program rather than the source code, our patches are more structured and contain extra information to transform the live state. Physically, patches are realised as a triple consisting of functions, data transformers, and a state transformer. The functions are either new functions not appearing in the original program, or functions that have changed between the versions. Data transformers are functions from the old to the new representation of a data type and support arbitrary changes to user defined types. Finally, the state transformer is an arbitrary procedure to be run after a patch has been applied, providing a way to manipulate the global state.

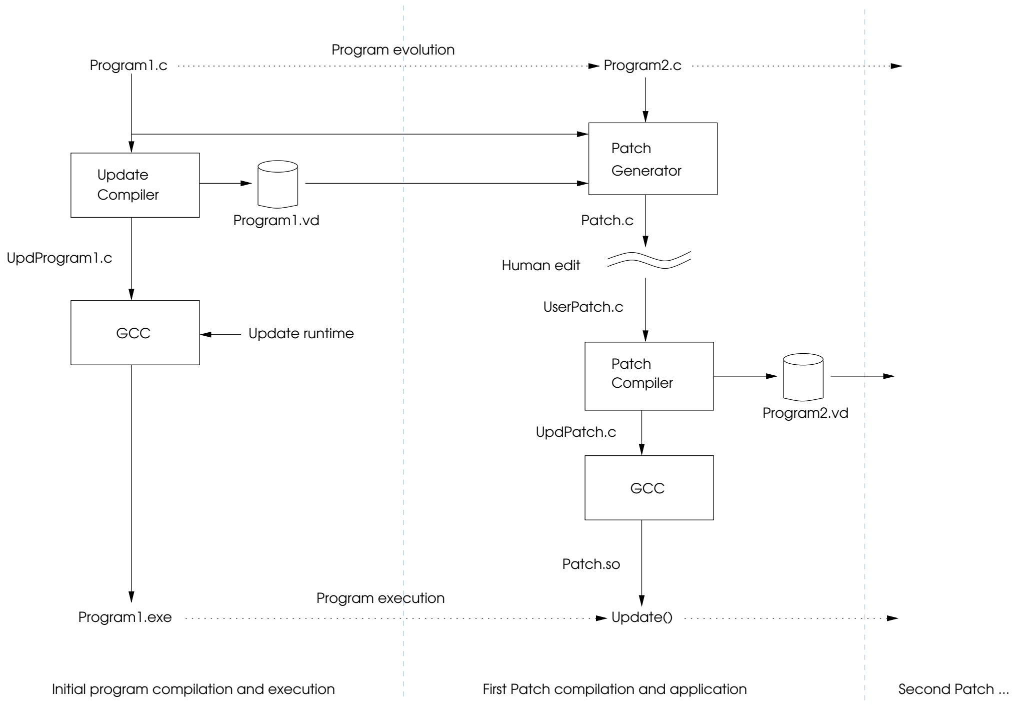

Figure 7.1: Update process overview

An overview of the evolution process is given in Figure 7.1, which we now explain. The developer writes their initial program, here `program1.c`, and transforms it with our update compiler. Using the resulting source code, an executable can be produced using a standard C compiler. The only requirement is that this program is linked with the

[4]See the Unix manual page for patch(1)

update runtime, a small library that handles requests to perform a dynamic update. The user then receives the executable, which he proceeds to run. Meanwhile, the developer continues to improve the program, eventually producing a new release, `program2.c`. His aim is to produce a *dynamic patch* which he can send to the user. To do this, he uses the patch generator. The patch generator looks at the differences between program1.c and program2.c and produces a patch source file which the developer then edits. The refined patch is then fed to the patch compiler that generates a dynamic library which, when loaded into the existing program by the update runtime system, will upgrade the program to the new version. Further updates can be produced and applied in the same manner.

7.1.1 An example

To demonstrate the utility of our approach we give a series of example updates to a linked list implementation. We show not only that it is possible to perform localised changes to the items stored in the linked list, but that transformations to the structure as a whole are also possible. The first update changes the items stored in the list, the second converts the linked list to a doubly linked list and the third makes use of the doubly linked structure introduced in the second update. We will refer back to this example when discussing compilation.

The source code in Figure 7.2 shows the implementation of a linked list (`struct Tlist`) that stores items of type `struct T`. Two list operations are shown: `concatT`, which prepends an item to a list, and `iterT`, which iterates a function over the items in a list. There is a single operation on items stored in the list, `printT`, which the long-running procedure `loop` iterates over its current list. The `main` function builds a list and calls `loop`.

Patch one The patch in Figure 7.3 demonstrates how to change `struct T` to contain an extra field. It defines three types: `T_old`, `T_new` and `T`. Here `T` is the new representation type, while `T_old` and `T_new` are the source and destination of the type transformer. The reader may wonder why we distinguish between the destination of the type transformer and the new type, after all they should be the same. The reason is technical. We use a special representation for data of an updatable user defined type, but the result of a type transformer function should remain in the standard representation. We use the two types to differentiate between them, treating `T_new` and `T` as incompatible, even through

```c
struct T { int x; };  /* Items to store in list */

/* Print a list item */
void printT(struct T* pt) {
   printf("t.x=%d\n", pt->x);
   return;
}

/* List node storing T items */
struct Tlist { struct T item; struct Tlist *next; };

/* Add an item to the list */
struct Tlist* concatT(struct T *pt, struct Tlist *pl) {
  struct Tlist *l = malloc(sizeof(struct Tlist));
  l->next = pl;
  l->item = *pt;
  return l;
}

/* iterate function f over list pl */
void iterT(void(*f)(struct T*), struct Tlist *pl) {
  while(pl) { f(&pl->item); pl = pl->next; }
  return;
}

/* Main program loop */
void loop(struct Tlist *pl) {
  iterT(printT,pl);

  getchar();
  __DSU_update(); /* Do an update */

  loop(pl); /* tail recursive call */
}

/* Initialisation */
int main() {
  struct Tlist *pl = 0;
  struct T t;
  int i;

  /* Create a list */
  for(i=0; i<10; i++) { t.x = i; pl = concatT(&t,pl); }

  loop(pl);
  return 0;
}
```

Figure 7.2: Source code for linked list

```
#pragma __DSU_TT("tt_T","struct_T","struct_T_old","struct_T_new")

struct T {
   int x;
   int y;
};

struct T_old {
   int x;
};

struct T_new {
   int x;
   int y;
};

void printT(struct T *pt) {
   printf("t.x=%d, t.y=%d\n", pt->x, pt->y);
   return;
}

void tt_T(struct T_old *xin , struct T_new *xout , struct T *xnew ) {
   xout->x = xin->x;
   xout->y = xin->x*2;
}
```

Figure 7.3: Patch one: change the items in the list to be a struct that stores two items

they both represent the updated type. The type `T` corresponds to the new abstract type, while `T_new` to its representation.

The type transformer `tt_T` converts `xin` to `xout`, initialising the extra field. The `xnew` argument is a pointer to the resulting `T` item after conversion; this will be used in patch two.

The names of the types and that of the type transformer are not fixed, but defined through a C pragma directive:

```
#pragma __DSU_TT("tt_T","struct T","struct T_old","struct T_new")
```

This associates the type `struct T` with its type transformer `tt_T` and the definition of the old and new types `struct T_old` and `struct T_new`.

Patch Two The patch in Figure 7.4 converts our singly linked list to a doubly linked one. The interesting point here is how the type transformer `tt_listT` works. Lines 20

```
#pragma __DSU_TT("tt_Tlist","struct_Tlist","struct_Tlist_old",
                 "struct_Tlist_new")

struct Tlist {
  struct T item;
  struct Tlist *next;
  struct Tlist *prev;
} ;

struct Tlist_old {
  struct T item;
  struct Tlist *next;
} ;

struct Tlist_new {
  struct T item;
  struct Tlist *next;
  struct Tlist *prev;
} ;

void tt_Tlist(struct Tlist_old *xin , struct Tlist_new *xout,
              struct Tlist *xnew ) {
  xout->item = xin->item;
  xout->next = xin->next;
  if(xin->next) xin->next->prev = xnew;
}

struct Tlist* concatT(struct T *pt, struct Tlist *pl) {
  struct Tlist *  l = malloc(sizeof(struct Tlist));
  l->next = pl;
  l->prev = (struct Tlist *)0;
  pl->prev = l;
  l->item = *pt;
  return l;
}
```

Figure 7.4: Patch two: change the list into a doubly linked list

```
void loop(struct Tlist *  pl ) {
  revIterT(printT, lastT(pl));

  getchar();
  __DSU_update();

  loop(pl);
}

void revIterT(void ( *f )(struct T *), struct Tlist *pl) {
  while(pl) {
    f(& pl->item);
    pl = pl->prev;
  }
  return;
}

struct Tlist* lastT(struct Tlist *pl) {
  while(pl->next) pl = pl->next;
  return pl;
}
```

Figure 7.5: Patch three: use the structure of the doubly linked list

and 21 copy the data as expected, but we do not see an initialisation of the `prev` field, as might be expected. The reason is that we have no way of knowing what the previous cell in the list is. Therefore, although the type transformer must act locally, we have to think globally. Here the solution is to set the next cells `prev` field while processing the current one. Using this strategy, the list will be converted as soon as all of the cells have been accessed concretely. We do not even need to force any accesses as the program will iterate over the list the next time it calls `loop`.

Patch Three The final patch, shown in Figure 7.5, adds two functions which make use of the reverse pointers added in the previous patch. `revIterT` iterates a function over a list in the order last to first and `lastT` returns the last element of a list. Finally, the main loop is modified to make use of these functions.

7.1.2 Tools

Before beginning a discussion of the individual stages involved in making programs updatable let us describe the tools used to create the compiler. Our implementation is written in the Objective CAML language [L$^+$01] using the CIL framework [NMRW02] to manipulate C source code and Banshee [Kod] to solve set constraints originating from the typing rules. We work with a type-annotated abstract syntax for C that CIL has simplified, where the key simplifications from raw C are: `switch` statements are converted to `if`; `for/while` statements are converted to `while(1)` using `break` and `continue`; local compound initialisers are expanded to assignments; and locally scoped variables are lifted to function scope.

7.2 Program compilation

The compilation follows the Proteus theory closely, but let us start by recalling the process and introducing the new stages required in the implementation.

Coercion insertion This is the first stage of compilation and inserts explicit coercions that witness the concrete and abstract use of structs, unions and typedefs.

Update analysis This closely follows the capability-based constraint analysis formally defined and proved correct in Chapter 6.

Layout This is a new stage required to deal with the low-level data representation. For example, user defined data types need to include space to expand in later updates and version information, while functions need to be indirected through lookup tables so that new ones can be swapped in.

Although the implementation is largely faithful to the theory, there is a timing difference in when instances of user defined types are updated. To recall, in the theory (Chapter 6) an update to type t results in occurrences of $\mathbf{abs}_{\mathrm{t}}e$ becoming $\mathbf{abs}_{\mathrm{t}}(c\ e)$, where c is a user supplied transformer from the old representation to the new. In the implementation we apply transformers at concretions instead of abstractions as it is more efficient. Why is it more efficient? Because applying coercions at abstraction points would require finding existing abstractions on the heap and stack, whereas applying them at concretions allows us to wait until we come across a named type value during computation and no explicit search is needed. We can feel comfortable with this change as the data is still converted before it is used concretely, thereby preserving representation consistency.

There is one consequence of this change that is worth pointing out. In the Proteus calculus $\mathbf{abs}_{\mathrm{t}}\ e$ always expects e to be an element of the latest definition of t. In the implementation this will not always be the case, rather abstractions will expect an argument of the same type as the definition of t at the time they were compiled. To see how this occurs assume that t = int and consider

$$\mathbf{let}\ x = \mathbf{abs}_\mathsf{t}\ 2\ \mathbf{in}\ \mathbf{let}\ \_ = \mathbf{update}\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{con}_\mathsf{t}\ (\mathbf{abs}_\mathsf{t}\ 3)\ \mathbf{in}\ e$$

The abstraction bound to x will expect an integer argument in both the calculus and the implementation. Now assume the update changes the definition of t from int to string supplying the conversion function c : int $\rightarrow$ string. In the calculus, **update** would change the expression bound to y to $\mathbf{con}_\mathsf{t}\ (\mathbf{abs}_\mathsf{t}\ (c\ 3))$ making the argument to the abstraction a string. As already mentioned, in the implementation conversion is delayed until a concretion, therefore c is not inserted and the abstraction still receives the integer 3. The implementation of the ensuing concretion then applies the conversion c.

7.2.1 Coercion insertion

As discussed in earlier chapters, the purpose of coercion insertion is to make the expressions of user defined type and their concrete use explicit. This extra information is then used in two ways. First in the analysis stage where the coercions indicate how the program depends on the representation of its data types, and later, at runtime, where the reduction of $\mathbf{con}_\mathsf{t} v$ is exactly the place to apply a user supplied coercion denoting an update to type t. The implementation introduces a pair of functions, $\mathbf{con}_\mathsf{t}$ and $\mathbf{abs}_\mathsf{t}$, for each user defined type t. We call these *witnessing coercions* and they have the following prototypes:

```
concrete_t  con_t(abstract_t *abs)
void  abs_t(concrete_t con, abstract_t *abs)
```

Here concrete_t and abstract_t represent the concrete and abstract forms of type t. For types defined using C's typedef facility the concrete and abstract types are obvious: given t = τ, t is abstract and τ is concrete. For composite types like structs there is no native concrete representation, thus for every struct (and union) type **struct** t we create a corresponding concrete type **struct** data_t, so that the two forms may be differentiated.

Coercion insertion takes place during a traversal of the program's abstract syntax. Concretions are inserted at struct and union field access, while a combination of concretions and abstractions are introduced whenever data of one type is used at another. This latter reclassification of data is directed by a subtype relation, as described in Chapter 6. We now comment further on these coercion insertions.

Composite types We start with an example: `x.l` is a concrete use of variable `x`, as we make the tacit assumption that it is a structure containing a field `l`. Given such an expression we witness the concrete use by sequencing the instruction $\mathbf{con}_{\mathtt{t}}(\mathtt{x})$ just before the field access.

Subtype relation At an assignment, the type of the data being assigned may not exactly match the type of the variable being assigned to. For example `t x` $= 3$ has an integer on the right hand side, but `x` is of type `t`. Assuming `t` is typedef'ed as an **int**, the subtype relation tells us that the resulting code should be `t x` $= \mathbf{abs}_{\mathtt{t}}\ 3$. This type of coercion also occurs for the arguments to functions.

Following the use of the $\lhd$ relation in the theory we must unfold the definition of named types where we require a particular structural type.

Type unsoundness

When inserting coercions we must deal with all the casts that C allows, but which are not type safe. If data of a user defined type falls into one of the following categories then we mark the type as non-updatable and refrain from inserting mediating coercions:

- Data of that type is cast to or from `void*`.
- The type is used in a non-trivial sizeof calculation.
- A named type that is the type of a bit field in a structure.
- A type that is defined externally to the program.
- Data of one type is used at another type and we cannot determine a valid subtype relation between them.

The most inconvenient of these is the first, although we believe that by using some standard techniques we could make our analysis more precise in the presence of `void*`. However, this is left to future work.

7.2.2 Update Analysis

Our implementation of the update analysis follows the theory presented in Chapter 6. This section discusses the implementation of the analysis which consists of a core constraint collection and solving phase with pre- and post-processing to help the user choose suitable update points.

Constraints Constraints are collected according to the rules given in Chapter 6, but extended to cope with extra constructs such as `goto` and `while` loops. Once the constraints are collected, the μ-constraints are solved and the solution used to resolve the conditional constraints (introduced by the call rule) into standard subset constraints on Δ sets. These Δ-constraints are solved using Banshee [Kod], a toolkit for building constraint-based analyses.

One interesting issue is what to do when we lose track of type information due to C's weak type system. More specifically, what we do when we lose track of function types, as this is the only place where type annotations occur. In this case we are forced to be conservative and constrain the function, forbidding it, or any function it calls, to perform an update. This guarantees that the function can be called from any context without affecting the validity of the analysis.

Speculative update insertion When we described the theory in Chapter 6 we assumed that update points were inserted by the user. In the implementation, we naturally still allow the user to do this. However, while the user must have ultimate control over the placement of updates, we would also like the compiler to assist in choosing suitable update points. In particular, we would like to know which points in our program could be *universal* update points, that is, a point in the program at which all types are free to be updated. To this end, *speculative* update points are inserted into the code at strategic points, e.g. before every `return` point, or after every block. After solving the constraints, we can determine which of the speculative update points are both universal and not defined in a function where the analysis ruled out updates. All the speculative update points except these valid universal ones can be deleted and the remaining ones presented to the user as candidate update points.

7.2.3 Layout

Up until now, we have looked at how the compiler analyses the updateability of a program. We now focus on how a program must be transformed to enable actual updates to take place. In order to update programs using the method we propose, it is necessary for the runtime to be able to replace functions and to check the version of a user defined data type before each concrete use [5]. To accomplish these two aims layout has two distinct phases: indirection of functions and wrapping of data structures.

[5] The reader worried about the efficiency of this can be reassured that it is not necessary to check the version if no update could have occurred since it was last checked. We have not implemented this optimisation in the current system, but it would be straightforward to add.

Function indirection

Function indirection introduces, for every function, a pointer to that function through which all calls are indirected. Consequently, updating one function for another is a local operation on the function pointer. Without this transformation all call sites would need altering when updating a function.

Data type wrapping

In this second phase, for every user defined data structure (e.g. `struct t` from our linked list example) we produce an auxiliary data structure that wraps the original one. The wrapper structure pairs storage for the actual data with a single integer representing the version number.

The version number records which version of the data type is stored in this wrapper. Every time a data type is upgraded, the version number is increased and variables of type `struct t` subsequently created by new code are given the this increased version number. Upon concretion, this number can be inspected to determine if a conversion needs to be applied.

In deciding how we store data of a user defined type, we must take into account that the size of the data stored here is likely to change. Thus, there is an interesting decision about how to store the data. There are two choices. The first is to use a fixed space, larger than the size of the initial data to be stored, to allow for later updates. While this has the advantage that stack allocated objects remain stack-allocated, it has the disadvantage that growth in the size of a data type is limited. The second option is to store in the wrapper a pointer to a heap location. Although this solution allows the size of the data type may grow arbitrarily, it requires an extra level of indirection at each access and allocation on the heap, which is more expensive than stack allocation. Moreover, memory management is a problem here: we would have to use a garbage collector, or derive a region-based allocation method to remove the heap allocated objects when they are no longer reachable. Garbage collection is usually unsuitable for programs written in C, although for the kind of long running server applications likely to benefit from DSU, the impact would be minimal. Our current implementation supports the fixed space approach, with the wrapper for `struct t` defined as follows:

```
union __DSU_udata_struct_T {
   struct __DSU_data_T __DSU_data;
   char __DSU_slop[max(2*sizeof(T),16U)];
};
```

```
struct __DSU_wrapper_struct_T {
   unsigned int __DSU_version;
   union __DSU_udata_struct_T __DSU_udata;
};
```

The first union provides space for the actual data extended with a `slop` that provides room for future updates to grow. The second struct wraps the union and provides version information.

Con and Abs Implementation

Now we have defined the data representation, we can discuss the implementation of coercions. Abstractions take a concrete data type and wrap it, while concretions do the reverse. They have the following definitions:

```
struct __DSU_data_T*
__DSU_con_struct_T(struct __DSU_wrapper_struct_T* abs )
{
  __DSU_latest_tt_struct_T(abs);
  abs->__DSU_version = __DSU_latest_type_version_struct_T;
  return (& abs->__DSU_udata.__DSU_data);
}

void __DSU_abs_struct_T(struct __DSU_data_T con ,
                        struct __DSU_wrapper_struct_T* abs )
{
  abs->__DSU_udata.__DSU_data = con;
  abs->DSU_version = 1;
}
```

Concretion applies the latest user type transformer (the identity function if no updates to this type have occurred), sets the data version to the most recent and returns a pointer to the concrete data structure. To abstract at a type `T`, the abstraction function is called, passing a concrete data structure to abstract and the address of a wrapper. The wrapper is then updated to contain the concrete data and the version number is set to the version of this abstraction function, i.e. to the number of times the function has been updated, here 1. Thus even after subsequent updates, this code may create instances of an abstract type at the old representation (as only code that concretises has to be replaced) but before the user sees it (i.e. upon concretion), it will be converted to the most recent representation.

7.2.4 Version Data

At the end of the initial program compilation and after every patch compilation, we must save information about user defined types, functions and global variables so that the validity of future patches can be determined.

For every user defined type we store the type definition, whether or not it is updatable and its version number. For every function we store its name, type, the name of the indirection variable that all calls go through and its version number. For global variables we store their definitions. All stored types contain the capability annotations which allows us to type check future updates according to the $\mathrm{updateOK}(-)$ predicate of the update capability analysis.

7.3 Abstraction-Violating Alias Analysis

While the Proteus theory does not deal with C's address-of operator, we must deal with this in any implementation targeting the full C language. In this section we discuss the problems posed by this operator and detail our approach to solving them.

To illustrate the problem the address-of operator, suppose type `t` is a struct with an integer field `i` and consider how coercions would be inserted:

$$\begin{array}{l} \textbf{struct}\ \mathtt{t}\ \mathtt{x};\\ \textbf{int} * \mathtt{n} = \&(\mathtt{x.i});\\ \textbf{update};\\ *\mathtt{n} = 42; \end{array} \quad \xrightarrow{\text{insert coercions}} \quad \begin{array}{l} \textbf{struct}\ \mathtt{t}\ \mathtt{x};\\ \textbf{con}_{\mathtt{t}}\mathtt{x};\\ \textbf{int} * \mathtt{n} = \&(\mathtt{x.i});\\ \textbf{update};\\ *\mathtt{n} = 42; \end{array}$$

The access to field `i`, a concrete use of `x`, is witnessed by a preceding concretion. However, the assignment on the last line also accesses field `i`, but this time *indirectly* through the pointer `n`. The second concrete use does not have an associated concretion. The issue here is that the address-of operator reveals information about the type `t`. Moreover, there is now a way to concretely use `x` that is disassociated from the type of `x`: we cannot determine on line 5 that assigning 42 to `*n` is a concrete use of structure `t`. We say that n is an *abstraction-violating alias* and that type t's abstraction is *violated*.

Whenever an abstraction-violating alias exists, there is the possibility of an undetected concrete use. Thus, an abstraction-violated type is not safe to update.

A conservative solution is to mark all violated structures as non-updatable. If some instance of a structure has an alias to one of its fields then that structure is marked as non-updatable.

A less conservative solution is to approximate the lifetime of abstraction-violating aliases. Outside the lifetime of such abstraction-violating aliases, updates to the structure are safe. If we could determine the set of structures whose fields might be aliased at each update point, or a conservative approximation of it, we could preclude these types from being updated only for that update point.

The practicality of the second option is supported by observing that many applications only take references to fields in order to pass the field's value to a function by reference, and that these references are not long lived. In our compiler we have implemented this as a simple constraint-based analysis, which we now describe informally.

7.3.1 Description

The analysis is constraint based. Sets of violated types are associated with pointers and functions. Constraints are gathered about the types that these sets should contain via a type-directed traversal of the program. These constraints are then solved to obtain the actual sets. Given these sets, we can calculate the types violated in the dynamic scope of each update point, and thus preclude them from update at that point.

With each pointer, we associate the set of types that that pointer may violate. In other words, the structures whose fields it may point at. Given a typing context, the union of all the abstraction violating sets associated with the pointer types it mentions, gives the set of types that are violated in that static environment.

With each function, we associate the set of types that may be abstraction-violated in the contexts from which it is called. These are the types violated in the dynamic environment of the function. Whenever a function is called, a constraint is placed on its associated abstraction-violation set so that it contains all of the types violated in the calling context's static environment.

For each update point, we form a conservative approximation of the set of types that may not be updated there. This is the union of the types violated in the static environment and those violated in the dynamic environment. These types are precluded from update at that point.

A formal description of the system will be published in future work.

7.4 Patch compilation

Having written a program, made that program updatable and run it, the next stage is to write, compile and apply a patch. In this section we look at the form of patches and how they are compiled into a dynamic patch.

As mentioned previously, user-written patches consist of new and replacement functions, together with specifications of types to update. As an example, recall the first patch to our linked list in Figure 7.3. The question now is how a file like this is transformed into a patch that can be applied by the runtime system? The first part of the answer is pleasingly simple: all functions and types are compiled in the same way as they were in program compilation. Two further stages are unique to patch generation. A second stage checks the validity of the patch, ensuring that for any types replaced, all functions that manipulate that type concretely are included in the patch, and that the replacing values are subtypes of the ones currently existing in the program. These checks are performed using the version data saved when we compiled the original program, or the last patch. A final stage produces code that installs the patch. Patch installation involves making sure both old and new code use the latest functions and that when the program comes across an old type that it converts it to the very latest version.

Type transformer integration When the user writes a type transformer, they produce a function to convert the current definition of a type to the new definition. After performing two updates to a type, it is possible that we will use some previously unused data from the original program which will require converting from version zero to one and then from one to two. Thus, type transformers need to be wrapped so that, depending on the version of the data to convert, they first invoke the previous type transformer.

Initialisation code This stage produces an initialisation procedure that installs the patch. This procedure acts as follows:

1. For each function in the patch, set that function's indirection variable to point to it.

2. For each type transformer in the patch, make the latest type transformer indirection variable point to the type transformer wrapper.

Program	KLoC	gcc (sec.)	Proteus-C (sec.)	μ	Δ
Apache 1.3.6	63	3.8	65	4,767	44,633
VSFTPD 2.0.2	25	1.2	14.6	1816	17,572
OpenSSH 4.0p1	85	4.8	88	88,5095	61,929

Table 7.1: Preliminary performance results for the Proteus-C compiler showing the application, the number of lines of code (1,000's), the time gcc takes to compile the program, the time that Cilly (the C parser and pretty printer we use) takes (including the final call to gcc, the number of μ-constraints, and the number of Δ-constraints.

7.4.1 Patch Generation

For large programs it is important to partially automate the generation of patches. Programs are updated in arbitrary ways, so completely automatic patch generation is obviously impossible and patches will always need to be edited by hand. We found a simple approach to patch generation to work best. Our patch generator simply compares the old and new versions of a program and produces a patch file that contains

1. all the user defined types and functions that have changed;
2. an empty type transformer shell for all of the changed types; and
3. all the new user defined types and functions

7.5 Performance

We tested our prototype implementation against Apache, VSFTPD and OpenSSH – three common Unix server applications. Our experience with our implementation is limited at the time of writing so we can not give detailed performance figures. This section gives some preliminary results.

The running times of the compiler in Table 7.1 show that the analysis scales well with an increase in the size of the code, even though compile times are roughly 15 times slower on average than gcc.

Interestingly, the server applications that we want to update are IO bound and so there is a negligible effect on overall performance. We have observed that the memory requirement of an updatable application is generally less than twice what the original application would have required. This is because the only significant space overhead is room to expand the user-defined data types.

We leave to future work reporting experience with updating real-life programs.

7.6 Patterns for Updatable Programs

Proteus provides a theoretical foundation for dynamic update, and the prototype implementation has gone some way to showing its practicality. By structuring the update mechanism on the existing user defined type facilities of C, we are able to take advantage of the structure already present in programs to perform dynamic update. Additionally, it allows existing C programs to be updated with minimal changes. However, updates can be made easier to write if programs are structured in an update-friendly way. In this section we present some design patterns that facilitate the updateability of programs; C has no abstraction mechanisms to enforce these design patterns, but following them will aid the update process.

Tail recursive calls instead of loops This is a necessary change. Long running loops are supported in our system through tail recursive calls. This is because old code that is active is executed to its end, but only new functions are called. An infinite while loop never leaves the current function, but a tail recursive loop will use the new code on its next iteration.[6]

Encapsulated access Type transformers must act locally, because the transformer only sees one data item at a time. If a data structure is made up of many sub-structures, such as our linked list example, then it can be useful to have a handle on the whole structure when wanting to convert it to another structure, like a tree. If all accesses are through an enclosing structure, then the outer structure will be updated before the inner structure is accessed. The outer structure can force the updates of the inner structures and in addition do the global processing on the structure.

Global handles Suppose you chose an update point in the middle of some tricky processing of a complicated data structure. In cases like these, it may be advantageous to convert all of the data being processed prior to the program resuming. To do this in our system we can simply iterate over the data making a concrete use of it in the state transformer, forcing the lazy conversion. In order to do this the state transformer function needs access to the data and so we need to provide a global handle.

[6]As an aside, it would be easy to adapt the update compiler to convert long running loops into tail-recursive functions by closure converting the body of the loop and lifting it into a new function.

7.7 Conclusions

This chapter presented an implementation of the Proteus theory presented in Chapter 6 for the full C programming language, discussed the design choices and hilighted areas where the implementation differed from the theory. We also gave an informal presentation of a new analysis to track abstraction-violating aliases and thus deal with the address-of operator in our framework. The implementation has validated the utility of our approach and through use we have gained a better understanding of how to write updatable programs.

8 Conclusions and Future Work

Dynamic update has traditionally been expressed at a low level of abstraction, unintuitive to the programmer and with few guarantees. This thesis raised the level of presentation to give a semantics at the language level, which is arguably more intuitive. It presented a theory of dynamic software updates dealing with dynamic rebinding, data transformation, and static determination of update capability. An exploration of the practicalities of the theory was initiated.

The rest of this chapter gives a summary of how this was achieved and discusses future directions for research.

8.1 Summary

We began by attempting to reconcile the semantics of the CBV λ-calculus with that of dynamic update. This lead to the discovery of the delayed instantiation calculi λ_r and λ_d. We then showed that the reasoning principles of the CBV λ calculus apply when thinking about λ_r and λ_d programs by showing that their contextual equivalence relations coincide. Within the delayed instantiation framework we showed how to express a dynamic update operation simply and directly. This foundational theory was then used to present a simple module-level update operation in the style of Erlang, and its strengths and weaknesses were demonstrated by example.

Having established a foundation for dynamic relinking, we turned to the challenge of state transformation. Up until this point, state transformation was carried out in an ad hoc manner, with no direct support for changing the representation of data. Part two of this thesis developed a theory of data type evolution that not only supports uniform changes to arbitrary data type representations, but amortises the cost of transforming instances across the running time of the program. While this system provided the required functionality and was type safe, it was unknown which types would be updatable when the program was ran; this could only be discovered when an update was attempted. This concern was addressed by providing a capability based type system that gives static guarantees about which types will be updatable at runtime.

The theory presented was then used to produce a prototype implementation for the C programming language. The implementation is capable of transforming C programs into updatable ones, performing the static analysis to give assurances on updatability, compiling dynamic patches, and applying them at runtime.

8.2 Future Work

There are many directions for future work. Here we limit our discussion to direct extensions of the work presented.

8.2.1 Integration with Other Language Features

In our work we have deliberately kept the number of language features low so that the problem remained tractable. Future work should investigate how dynamic update interacts with other language features. For example, consider extending Proteus and $\mathrm{Proteus}^{\Delta}$ with modern programming language constructs. Proteus could be augmented with most type system extensions (e.g. parametric polymorphism) in a straight-forward way. One area which requires consideration is the addition of first class functions. Due to our requirement that all functions that use a named type concretely are updated along with the type, intricacies arise with anonymous functions and partial applications. Anonymous functions are used as a low-level building block in functional programs and there may be many of them, for example, if a state monad is implemented in ML. In this situation it is unclear that the requirement of function replacement mentioned earlier is still a reasonable one.

8.2.2 Concurrency

Although a lot of server software (especially under Unix) is process based, it is a growing trend to be thread based instead. The theories of dynamic rebinding and delayed instantiation extend to accommodate threads in a straight-forward way. It is the state transformation of Proteus that is more challenging when extending to a threaded environment. When a thread reaches an update point it would be unable to carry out a confree check to guarantee the safety of the update because this check would depend on the state of the other threads, which would be constantly changing. The obvious option is to take a barrier approach, waiting for *all* threads to reach an update point. Given such a system state, as long as **UpdateOK**$(-)$ held in one thread and the confree check passed at all threads, the update would be safe to apply.

Similarly, the capability based type system needs extending to cope with threads. The property it approximates is flow sensitive, but with threads the flow is non-deterministically interleaved between concurrent paths of execution. Below some paths of inquiry are suggested, but more exploration is needed.

Barrier approach Following the suggestion made in the dynamic case, all threads could be required to reach an update point before an update could proceed. As each update point is annotated with the named types that *may not* be updated there we have a set of n such restricting sets, say $D = \Delta_1, \Delta_2, \ldots, \Delta_n$. The safe types to update are those that are safe to update at each update point, i.e. $\bigcap D$.

Separation approach This is a refinement on the barrier approach. Each thread in a program usually has a specific job, such as storage access or handling a request, and is started from a specific procedure. Associate with each thread a *sort* indicating the function where execution begins. The code that can be reached from a given starting point is limited and therefore, it is possible to determine an approximation to the functions that are callable from that thread. Thus, we could determine an approximation for the types concreted by each thread sort. When updating a named type we only need synchronise the threads that actually concrete values of that type.

8.2.3 Implementation

An implementation is an important part of validating the practicality of theoretical ideas. Although a working prototype has been produced that demonstrates the idea's feasibility, we clearly need to test our ideas with a larger scale study of industrial-sized applications. Work is currently under way, in collaboration with colleagues at the University of Maryland, to produce patches to upgrade the Very Secure FTP Daemon (VSFTPD) and Secure Shell Daemon (SSHD) that will match their source code evolution over past few years. Initial results are very encouraging.

From a technical point of view, the issues that need addressing in the current implementation for C mainly concern the analysis. One such issue is `void *` pointers. We are currently very conservative in dealing with pointers of type `void *`. Using techniques from the literature for dealing with parametric and existential uses of `void *` would enhance the flexibility of the analysis.

We have a PLDI publication [NHSO06] that details our experience of using Proteus to update industrial network server applications written in C.

Nomenclature

$\lvert a \rvert$	The maximum label in the labelled term a
$\Phi \blacktriangleleft$	Well-formed environment
$\Phi \blacktriangleleft a$	Environment well-formed w.r.t. the term a
$\Phi \blacktriangleleft E$	Environment well-formed w.r.t. the evaluation context E
$\longrightarrow$	Single step reduction relation
$\longrightarrow^*$	Transitive, reflexive closure of $\longrightarrow$
$\_$	Hole in a context
$\xrightarrow{\mu;\Delta}$	An update reduction in Proteus-Delta
$[\![-]\!]_{\mathsf{val}}$	Value collapsing function converting $\lambda_{r/d}$ values to λ_c values
$\{v/x\}e$	The substitution of v for x in expression e
A	Atomic evaluation context
A_1	Non-binding atomic evaluation context
A_2	Binding atomic evaluation context
a	An annotated term
$\mathrm{bindOK}[\,H\,]^{\mathrm{upd}}$	Predicate to determine if update upd is compatible with heap H
$\mathcal{C}$	A coercion context
$\mathbb{C}$	Program context
$\mathcal{E}_c[E_3]^{\Phi}$	The environment implied by E_3 and Φ

e cval	λ_c value predicate
Δ	Capability Set (of named types)
a d'val	$\lambda_{d'}$ value predicate
a dval	λ_d value predicate
E	General evaluation context
e	A term of the calculus under consideration
$E.E'$	Context Composition
$E.e$	Context application
$E[e]$	Context application
E_1	Non-binding compound evaluation context
E_2	Binding only compound evaluation context
E_3	Binding and non-binding evaluation context
$\mathrm{env}(E)$	The typing context generated by E
ϵ	Variable ranging over updatability flags
$\epsilon[a]$	Erase annotation on term a
e err	Indicates that e is stuck (a runtime error)
$\mathrm{frf}(a)$	The set of free recursive functions in a
$\mathrm{fv}(e)$	The set of free variables in e
$\Gamma \vdash e : \tau$	Typing judgement: e has type τ in context Γ
Γ	A typing context mapping variables to their types (and possibly type names to types)
$\Gamma, x : \tau$	Disjoint extension of Γ with $x : \tau$
$\Gamma[x : \tau]$	The context Γ, but with x mapping to τ
Γ, Γ'	The context formed by the disjoint union of the elements in Γ and Γ'

H	Heap
$\mathrm{hb}(E)$	The set of variables that bind around the hole in E
$a\ \mathsf{inf_d}$	Predicate indicating that a is in instantiation normal form for λ_d
$a\ \mathsf{inf_d^\circ}$	Predicate indicating that a is in open instantiation normal form λ_d
$a\ \mathsf{inf_r}$	Predicate indicating that a is in instantiation normal formfor λ_r
$a\ \mathsf{inf_r^\circ}$	Predicate indicating that a is in open instantiation normal form λ_r
$\iota[e]$	Create annotated term from unannotated
$\mathrm{instvar}[a]$	The number of variables above lambda abstractions
$\mathrm{label}_l(a)$	The term a labelled with natural numbers
μ	Updatability flag
N	Not updatable value for updatability flag
Ω	Type environment
ϕ	Variable ranging over capability sets
R	Destruct context or relation over terms
r	Reference cell
$\mathrm{rebind}(V, L)$	The substitution replacing the variables in V with their (unique) alpha variant in L
$a\ \mathsf{r'val}$	$\lambda_{r'}$ value predicate
$a\ \mathsf{rval}$	λ_r value predicate
Σ	Module context
$\Sigma; \Gamma \vdash e : T$	Expression e has type T in typing context Γ and module context Σ
t	A named type
P	A program
τ	A Type

U	Updatable value for updatability flag
upd	Proteus update
$\mathrm{updateOK}(\mathrm{Upd}, \Omega, H, e)$	UpdateOK predicate
v	A value
$\mathrm{wf}[-]$	Well-formed predicate
x, y, z	Variables
$a\ \ \mathsf{znf}_{\mathsf{d}}$	Predicate indicating that a is in zero normal form for λ_d
$a\ \ \mathsf{znf}_{\mathsf{d}}^{\circ}$	Predicate indicating that a is in open zero normal form λ_d
$a\ \ \mathsf{znf}_{\mathsf{r}}$	Predicate indicating that a is in zero normal form for λ_r
$a\ \ \mathsf{znf}_{\mathsf{r}}$	Predicate indicating that a is in open zero normal form λ_r

Appendices

Appendix

A

Proof of Type Soundness for Proteus

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

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

A.0.1 Lemma (Progress). *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Omega, \Phi \vdash e : \tau$ *then either*

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

(ii) e *is a value*

A.0.2 Lemma (Preservation). *If* $\emptyset \vdash \Omega; H; e : \tau$ *then*

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

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

A.0.3 Theorem (Type Soundness). *If* $\emptyset \vdash \Omega; H; e : \tau$ *then either*

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

(ii) e *is a value*

Proteus' type system is a simplification of that of Proteus^{Δ}. To obtain Proteus, the capabilities are removed, along with the subtype relation and the subsumption rule.

Subtyping can be removed as their only function was to provide subtype-polymorphic behaviour for the capabilities on function arrow, which do not exist in the dynamic system.

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

A.0.4 Lemma ($\mathrm{conFree}[-]$ Congruence). *For any* upd *and* e, *if* $\mathrm{conFree}[\,e\,]^{\mathrm{upd}}$ *holds, then for any subterm* e' *of* e *we have* $\mathrm{conFree}[\,e'\,]^{\mathrm{upd}}$. *We say that* $\mathrm{conFree}[-]^{\mathrm{upd}}$ *is congruent to the syntax of expressions.*

A.0.5 Lemma ($\mathrm{updateOK}(-)$ Congruence). *For any* upd, Ω, H *and* e, *if* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e)$ *holds then for any subterm* e' *of* e *we have* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e')$. *We say that* $\mathrm{updateOK}(-\mathrm{upd}, \Omega, H, -)$ *is a congruent to the syntax of expressions.*

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

A.0.6 Lemma (Update Expression Safety). *If* $\vdash\ \Omega$ *and* $\Omega;\Phi\ \vdash\ H$ *and* $\Omega,\Phi,\Gamma \vdash_\mu e : \tau$ *and* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e)$ *then* $\mathcal{U}[\Omega,\Phi,\Gamma]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[e]^{\mathrm{upd}} : \tau$

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

➤*Case* (EXPR.XVAR) **:** By assumption $\Omega,\Phi,\Gamma \vdash \mathsf{z} : \tau$. Thus $\Phi(\mathsf{z}) = \tau$ and by (a) $\mathsf{z} \in \mathrm{dom}(H)$.
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on expressions we have $\mathcal{U}[\mathsf{z}]^{\mathrm{upd}} = \mathsf{z}$.
There are three ways in which $\mathcal{U}[\Omega,\Phi,\Gamma]^{\mathrm{upd}}(\mathsf{z}) = \tau'$ can arise:

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$ **:** Let $\mathrm{upd.UB}(\mathsf{z}) = (\tau', b_v)$.

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

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

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

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

➤*Case* (EXPR.ABS) **:** By assumption

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

Consider the form of upd.UN:

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

➤*Case* $\mathrm{upd.UN}(\mathsf{t}) = (\tau'', \mathsf{c})$ **:** Observe $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}})$. Using (A.EXPR.ABS) and (A.EXPR.APP) we are required to prove:

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

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

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

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

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

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

➤*Case* (EXPR.CON) **:** Assume

$$\frac{\Omega(\mathsf{t}) = \tau \qquad \Omega, \Phi, \Gamma \vdash e : \mathsf{t}}{\Omega, \Phi, \Gamma \vdash \mathbf{con}_{\mathsf{t}}\ e : \tau} \tag{A.1}$$

$$\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \mathbf{con}_{\mathsf{t}}\ e) \tag{A.2}$$

Suffices to show that the leaves of this derivation hold:

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

By updateOK assumption: $\mathrm{conFree}[\,\mathbf{con}_{\mathsf{t}}\ e\,]^{\mathrm{upd}}$, thus by definition of conFree, $\mathsf{t} \notin \mathrm{dom}(\mathrm{upd.UN})$. It follows by definition of $\mathcal{U}[-]^{\mathrm{upd}}$ that (a) holds.

By Confree Congruence lemma $\mathrm{conFree}[\,e\,]^{\mathrm{upd}}$. It can now be shown by application of IH that (b) holds.

❑

A.0.7 Lemma (Heap Update Safety). *If* $\vdash\ \Omega$ *and* $\Omega;\Phi\ \vdash\ H$ *and* $\mathrm{updateOK}(\mathrm{upd},\Omega,H,e)$ *then* $\mathcal{U}[\Omega]^{\mathrm{upd}};\mathcal{U}[\Phi]^{\mathrm{upd}}\ \vdash\ \mathcal{U}[H]^{\mathrm{upd}}$

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

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

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

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

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

So assume (a)-(d) and also

$$\vdash\ \Omega \tag{A.3}$$

$$\mathrm{updateOK}(\mathrm{upd},\Omega,H,\hat{\Delta}) \tag{A.4}$$

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

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

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

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

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

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

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

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

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

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

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

➤*Case* (iii) **:** Assume

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

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

$$\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \ \vdash \ \mathcal{U}[\lambda(x).e\ ]^{\mathrm{upd}} : \tau_1 \rightarrow \tau_2 \tag{A.5}$$

$$\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}}(\mathsf{z}) = \tau_1 \rightarrow \tau_2 \tag{A.6}$$

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$ **:** This case is dealt with as in the $\mathrm{Proteus}^{\Delta}$case as it only relies on properties of updateOK common between the two definitions.

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$ **:** By case split and definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on heaps, there exists b', H' such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\mathrm{upd}} = z \mapsto (\tau, \mathcal{U}[b']^{\mathrm{upd}}), \mathcal{U}[H']^{\mathrm{upd}}$ and $H = \mathsf{z} \mapsto (\tau, b'), H'$.

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

By (c) and typing rules

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

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

Required to prove A.5 and A.6.

By B.5 $\mathrm{conFree}[\,H\,]^{\mathrm{upd}}$, therefore $\mathrm{conFree}[\,e\,]^{\mathrm{upd}}$. By UpdateOK Congruence lemma $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e)$.

By Update Expression Safety lemma $\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[e]^{\mathrm{upd}} : \tau_2$. Therefore, by use of (A.BIND.FUN), A.5 holds.

By the definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts it follows that $\mathcal{U}[\Phi]^{\mathrm{upd}}(z) = \tau$ making A.6 holds, as required.

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

❑

Appendix

B

Proof of Type Soundness for Proteus-Delta

In this section we give a proof of type soundness for $\mathrm{Proteus}^{\Delta}$. The proof is based on standard techniques for proving syntactic type soundness [WF94], but extended to deal with a capability type system. Proving soundness for the system guarantees that our inferred update points are safe and consequently any update judged suitable by $\mathrm{updateOK}()$ at runtime will not invalidate the type of the program.

Before we begin the proof we discuss the rôle of update capability weakening (Lemma B.1.13), a key lemma. Update capability weakening shows that the Δ annotation on updates is faithful to our intended meaning, that is, given an update point $\mathbf{update}^{\hat{\Delta}}$ in redex position in some larger expression, the only types used concretely following the update are contained in $\hat{\Delta}$. More formally, if

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

then

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

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

B.1 Proof of Type Soundness

B.1.1 Definition (Typing contexts). A context Γ is a finite partial function with the following entries:

$\mathsf{z} : \tau$	types of external identifiers
$\mathsf{z} : \tau\ \mathbf{ref}$	types of references
$x : \tau$	types of local identifers
$\mathsf{t} = \tau$	named type definitions

We write Φ for typing contexts containing only external identifiers and references, and Ω for those containing only named type definitions. ❑

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

B.1.2 Lemma (Inversion – expressions).

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

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

❑

B.1.3 Lemma (Inversion – subtyping).

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

2. *If* $\tau <: \{\mathrm{l}_1 : \tau_1, \dots, \mathrm{l}_n : \tau_n\}$ *then* $\exists \tau'_1, \dots, \tau'_n$ *such that* $\tau = \{\mathrm{l}_1 : \tau'_1, \dots, \mathrm{l}_n : \tau'_n\}$

❑

B.1.4 Lemma (Canonical Forms).

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

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

3. *If* v *is a value of type* $\{\mathrm{l}_1 : \tau_1, \dots, \mathrm{l}_n : \tau_n\}$ *then* $\exists v_1, \dots, v_n$ *such that* $v = \{\mathrm{l}_1 = v_1, ..., \mathrm{l}_n = v_n\}$

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

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

❑

B.1.5 Lemma (Weakening). *If $\Delta_1; \Gamma \vdash_\mu e : \tau; \Delta_2$ and $\Gamma \subseteq \Gamma'$ then $\Delta_1; \Gamma' \vdash_\mu e : \tau; \Delta_2$* ❑

We now establish some basic facts about capabilities.

- If an update is judged safe for one update point then it is also safe for any more restricted update point.
- If an expression does not perform an update, it does not consume any of its capability. By "consume" we mean that its pre capability is equal to its post capability.
- If a term type checks given one capability the it type checks in a larger capability.
- Values type with any pre and post capability (as long as pre is as least as permissive as post)

B.1.6 Lemma (UpdateOK Capability Weakening). *If $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta)$ and $\Delta' \subseteq \Delta$ then $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta')$*

Proof. The only clause in $\mathrm{updateOK}()$ that depends on Δ is $\mathrm{dom}(\Delta) \cap \mathrm{dom}(\mathrm{upd.UN}) = \emptyset$, and the validity is uneffected by the shrinking of Δ. ❑

B.1.7 Lemma (Capability Strengthening).

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

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

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

➤*Case* (A.EXPR.APP) : Assume

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

prove

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

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

➤*Case* (A.EXPR.UPDATE) : This is trivially true as type checking update requires U annotation on the turnstyle.
The rest of the cases are similar.

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

B.1.8 Lemma (Value Typing). *If* $\Delta_1; \Gamma \vdash_\mu v : \tau; \Delta_2$ *then* $\forall \Delta'_1, \Delta'_2, \mu'$ *such that* $\Delta'_2 \subseteq \Delta'_1$ *it holds that* $\Delta'_1; \Gamma \vdash_{\mu'} v : \tau; \Delta'_2$

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

➤*Case* (A.EXPR.INT) : By (A.EXPR.INT) $\Delta'_2; \Gamma \vdash_\mu n : \mathsf{int}; \Delta'_2$. By Capability Strengthening Lemma $\Delta'_1; \Gamma \vdash_\mu n : \mathsf{int}; \Delta'_2$ as required.

➤*Case* (A.EXPR.XVAR) : Similar to (A.EXPR.INT) case.

➤*Case* (A.EXPR.RECORD) : By Lemma B.1.4 (Canonical Forms) each element of the record must be a value in order for the record to be a value. The result follows by induction on each of the elements of the record and use of the (A.EXPR.RECORD) rule.

➤*Case* (A.EXPR.ABS) : By Lemma B.1.4 (Canonical Forms) of values $\mathbf{abs}_t$ is a value only if e is a value, thus suppose $e = v$ for some v, then the result follows by induction on the typing derivation of v and use of the (A.EXPR.ABS) rule.

➤*Case* (A.EXPR.SUB) : By straight-forward application of the IH.

❑

B.1.9 Lemma (Substitution). *If* $\Delta_1; \Gamma, x : \tau' \vdash_\mu e : \tau; \Delta_2$ *and* $\Delta_3; \Gamma \vdash_\mu v : \tau'; \Delta_3$ *then* $\Delta_1; \Gamma \vdash_\mu e[v/x] : \tau; \Delta_2$

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

B.1.10 Lemma (Derivations have Well-Formed Types). *If* $\Delta_1; \Gamma, \Omega, \Phi \vdash_\mu e : \tau; \Delta_2$ *then* $\Omega \vdash \tau$

B.1.11 Lemma (Typing Weakens Capability). *If* $\Delta_1; \Gamma \vdash_\mu e : \tau; \Delta_2$ *then* $\Delta_2 \subseteq \Delta_1$

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

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

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

This lemma is a key component in proving Update Capability Weakening (Lemma B.1.13).

B.1.12 Lemma ($\mathbb{E}$-inversion). *If* $\Delta; \Gamma \vdash_\mu \mathbb{E}[e] : \tau; \Delta'$ *then there exists* $\hat{\Delta}' \supseteq \Delta'$ *and* τ' *such that*

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

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

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

➤*Case* (A.EXPR.VAR) **:** In this case $\mathbb{E} = \_$, $e = x$ and $\Delta' = \Delta$.

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

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

➤*Case* (A.EXPR.INT|XVAR|UPDATE) **:** These cases all follow in a similar way to the (A.EXPR.VAR) case.

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

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

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

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

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

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

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

➤*Case* (A.EXPR.APP) **:** There are two possibilities for the form of $\mathbb{E}$: $v\ \mathbb{E}'$ and $\mathbb{E}'\ e$. We just consider the first as the second is similar.

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

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

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

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

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

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

Thus, it suffices to prove

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

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

➤*Case* (A.EXPR.CON) **:** $\mathbb{E} = \mathbf{con}_\mathsf{t}$. Assume

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

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

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

and prove

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

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

➤*Case* (A.EXPR.IF) **:** There are two cases for the form of $\mathbb{E}$: **if** $v = \mathbb{E}'$ **then** e_1**else** e_2 and **if** $\mathbb{E}' = e$ **then** e_1**else** e_2. We consider only the first as the second is similar.

Assume $\mathbb{E} =$ **if** $\mathbb{E}' = e$ **then** e_1**else** e_2 and

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

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

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

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

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

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

❑

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

B.1.13 Lemma (Update Capability Weakening). *If* $\Delta;\Gamma \vdash_\mu \mathbb{E}[\textbf{update}^{\Delta''}] : \tau;\Delta'$ *then* $\Delta'';\Gamma \vdash_\mu \mathbb{E}[\textbf{update}^{\Delta''}] : \tau;\Delta'$.

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

B.1.14 Lemma (Heap Extension)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\emptyset; \Omega, \Phi \vdash_{\mathsf{N}} v : \tau; \emptyset$ *and* $l \notin \mathrm{dom}(H)$ *then* $\Omega; \Phi, l : \tau\ \mathbf{ref} \vdash H, l \mapsto (\cdot, v)$

Proof. By definition of heap typing. ❑

B.1.15 Lemma (Update Expression Safety)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Delta_1; \Omega, \Phi, \Gamma \vdash_\mu e : \tau; \Delta_2$ *and* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta_1)$ *then* $\Delta_1; \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[e]^{\mathrm{upd}} : \tau; \Delta_2$

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

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

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

➤*Case* (A.EXPR.XVAR) **:** By assumption $\Delta; \Omega, \Phi, \Gamma \vdash_\mu \mathsf{z} : \tau; \Delta'$. Thus $\Phi(\mathsf{z}) = \tau$ and by (a) $\mathsf{z} \in \mathrm{dom}(H)$.
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on expressions we have $\mathcal{U}[\mathsf{z}]^{\mathrm{upd}} = \mathsf{z}$.
There are three ways in which $\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}}(\mathsf{z}) = \tau'$ can arise:

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$ **:** Let $\mathrm{upd.UB}(\mathsf{z}) = (\tau', b_v)$.

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

By updateOK assumption $\mathcal{U}[\Omega, \mathrm{types}(H)]^{\mathrm{upd}} \vdash \mathrm{heapType}(\tau', b_v) <: \tau$.

By Weakening $\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash \mathrm{heapType}(\tau', b_v) <: \tau$.

The result follows by use of (A.EXPR.SUB) type rule.

➤*Case* $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$ **:** By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \tau$, thus $\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash_\mu \mathsf{z} : \tau; \Delta'$, as required.

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

➤*Case* (A.EXPR.ABS) **:** By assumption

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

Consider the form of upd.UN:

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

➤*Case* $\mathrm{upd.UN}(\mathsf{t}) = (\tau'', \mathsf{c})$ **:** Observe $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}})$. Using (A.EXPR.ABS) and (A.EXPR.APP) we are required to prove:

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

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

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

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

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

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

➤*Case* (A.EXPR.CON) : Assume

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

$$\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta) \tag{B.3}$$

Suffices to show that the leaves of this derivation hold:

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

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

$$\begin{array}{lr} \Omega = \mathsf{t} = \tau, \Omega' & \text{for some } \Omega', \text{by B.3} \\ \mathrm{dom}(\Delta') \cap \mathrm{dom}(\mathrm{upd.UN}) = \emptyset & \text{by B.3} \\ \mathcal{U}[\mathsf{t} = \tau, \Omega']^{\mathrm{upd}} = (\mathsf{t} = \tau, \mathcal{U}[\Omega']^{\mathrm{upd}}) & \text{as } \mathsf{t} \notin \mathrm{upd.UN} \end{array}$$

➤*Case* (A.EXPR.RECORD) : By assumption (where $\Delta \equiv \Delta_0, \Delta' \equiv \Delta_n$):

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

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

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

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

➤*Case* (A.EXPR.SUB) : Assume

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

Prove

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

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

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

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

❑

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

B.1.16 Lemma (Heap Update Safety). *If* $\vdash \Omega$ *and* $\Omega;\Phi \vdash H$ *and* $\text{updateOK}(\text{upd},\Omega,H,\hat{\Delta})$ *then* $\mathcal{U}[\Omega]^{\text{upd}};\mathcal{U}[\Phi]^{\text{upd}} \;\vdash\; \mathcal{U}[H]^{\text{upd}}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

By (d) there exists a τ' such that

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

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

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

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

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

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

By Update Expression Safety lemma:

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

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

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

➤*Case* (ii) **:** Assume $\mathcal{U}[H]^{\mathrm{upd}}(z) = (\tau, e)$ and prove

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

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

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathrm{upd.UB}(\mathsf{z} = (\tau, e)$ **:** As (a) holds by assumption and $\mathsf{z} \in \mathrm{dom}(H)$ by case split, we have $\mathsf{z} \in \mathrm{dom}(\Phi)$. Thus for some τ', $\Phi(\mathsf{z}) = \tau'$.
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \tau\ \mathbf{ref}$, which proves B.11.
By B.9 $\mathcal{U}[\Omega, \mathrm{types}(H)]^{\mathrm{upd}} \vdash e : \tau$.
By context weakening $\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash e : \tau$, which proves B.10.

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$ **:** By the definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on heaps there must exist b' such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\mathrm{upd}} = \mathsf{z} \mapsto (\tau, \mathcal{U}[b']^{\mathrm{upd}}), \mathcal{U}[H']^{\mathrm{upd}}$.
As $b = \mathcal{U}[b']^{\mathrm{upd}}$ and this is an expression by fact we are in (ii) case split, b' must also be an expression. Thus by (b) $\Omega, \Phi \vdash b' : \tau$ and $\Phi(\mathsf{z}) = \tau\ \mathbf{ref}$. Then by inversion $\emptyset; \Omega, \Phi \vdash_{\mathsf{N}} b' : \tau; \emptyset$ holds.
By Update Expression Safety lemma

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

which proves B.10
As $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$, by the definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts we have $\mathcal{U}[\Omega, \Phi', \mathsf{z} : \tau\ \mathbf{ref}]^{\mathrm{upd}} = \mathsf{z} : \tau\ \mathbf{ref}, \mathcal{U}[\Omega, \Phi']^{\mathrm{upd}}$, which proves B.11.

➤*Case* $\mathsf{z} \notin \mathrm{dom}(H)$ **:** In this case it must hold that $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.AB})$.
By assumption$\mathcal{U}[H]^{\mathrm{upd}}(\mathsf{z}) = (\tau, e)$ (where e is in fact a value) therefore $\mathrm{upd.AB}(\mathsf{z}) = (\tau, e)$.
By B.9

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

which proves B.10.

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

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

which proves B.11, as required.

➤*Case* (iii) : Assume

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

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

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

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

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$: By case split there exists $x, e, \tau_1, \tau_2, \mu', \Delta_1, \Delta_2$ such that $\mathrm{upd.UB}(\mathsf{z}) = (\tau_1 \xrightarrow{\mu';\Delta_2} \tau_2, \lambda^{\Delta_1}(x).e\ )$.

$$\begin{array}{lr}
\mathcal{U}[\Omega, \mathrm{types}(H)]^{\mathrm{upd}} \ \vdash\ \lambda^{\Delta_1}(x).e\ : \tau_1 \xrightarrow{\mu';\Delta_2} \tau_2 & \text{by B.9} \\
\mathcal{U}[\Omega, \mathrm{types}(()H)]^{\mathrm{upd}} \ \vdash\ \lambda^{\Delta_1}(x).e\ : \tau_1 \xrightarrow{\mu';\Delta_2} \tau_2 & \text{by weakening}
\end{array}$$

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

➤*Case* $\mathsf{z} \in \mathrm{dom}(H)$ and $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$: By case split and definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on heaps, there exists b', H' such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\mathrm{upd}} = z \mapsto (\tau, \mathcal{U}[b']^{\mathrm{upd}}), \mathcal{U}[H']^{\mathrm{upd}}$ and $H = \mathsf{z} \mapsto (\tau, b'), H'$.

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

By (c) and typing rules

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

where $\tau = \tau_1 \xrightarrow{\mu;\Delta'} \tau_2$.

Required to prove B.12 and B.13.

By B.9 $\mathrm{bindOK}[\,\mathrm{types}(H)\,]$. By case split $\mathrm{z} \notin \mathrm{dom}(\mathrm{upd.UB})$. By last two facts $\mathrm{dom}(\mathrm{upd.UN}) \cap \Delta = \emptyset$. It follows by th eprevious fact B.9 that $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta)$.

By Update Expression Safety lemma $\Delta;\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash_{\mu} \mathcal{U}[e]^{\mathrm{upd}} : \tau_2; \Delta'$. Therefore, by use of (A.BIND.FUN), B.12 holds.

By the definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts it follows that $\mathcal{U}[\Phi]^{\mathrm{upd}}(z) = \tau$ making B.13 holds, as required.

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

❑

B.1.17 Lemma (Update Program Safety)**.** *If*

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

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

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

Proof. Assume

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

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

It suffices to prove the hypotheses for this deduction:

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

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

By Update Capability Weakening lemma and B.14

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

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

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

B.1.18 Lemma (Non-Update Expression Safety). *If* $\vdash \ \Omega$ *and* $\Omega; \Phi \ \vdash \ H$ *and* $\Delta_1; \Omega, \Phi \vdash_\mu e : \tau; \Delta_2$ *and* $H; e \ \longrightarrow \ H'; e'$ *then* $\exists \Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ *such that*

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

(ii) $\Delta_1'; \Omega, \Phi' \vdash_\mu e' : \tau; \Delta_2'$

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

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

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

➤*Case* (A.TYPE.APPU) : Assume

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

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

$$H; e_1 \ e_2 \ \longrightarrow \ H'; e' \tag{B.18}$$

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

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

Take $H' = H$ and $\Omega' = \Omega$ then (i) holds by B.16.

Now prove (ii) by showing

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

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

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

From typing of the LHS of the application

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

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

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

and thus

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

By use of subsumption on B.20 using facts from B.23

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

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

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

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

➤*Case* (A.EXPR.CON) **:** Assume

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

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

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

➤*Case* (A.EXPR.PROJ) **:** Assume

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

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

The only possible expression reduction is (PROJ). Assume

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

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

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

as required to show (b).

➤*Case* (A.EXPR.LET) **:** Assume

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

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

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

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

➤*Case* (A.EXPR.UPDATE) **:**

The case is trivially true as the expression cannot do a non-update reduction.

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

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

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

The only reduction rule applicable here is ref. This implies, for some value v and location $r \notin \mathrm{dom}(H)$, that $e'' = v$ and $H' = H, r \mapsto v$.

Required to prove

(i) $\Omega; \Phi, r : \tau\ \mathbf{ref} \vdash H, r \mapsto (\cdot, v)$

(ii) $\Delta; \Omega, \Phi, r : \tau\ \mathbf{ref} \vdash_\mu r : \tau\ \mathbf{ref}; \Delta'$

(i) follows from Heap Extension lemma. (ii) is deducible from (A.EXPR.LOC) and (A.EXPR.SUB).

➤*Case* (A.EXPR.(IF | DEREF | ASSIGN | ABS | IF-UPDATE)) **:** These cases follow similarly.

❑

B.1.19 Lemma (Non-Update Program Safety). *If* $\vdash \Omega$, $\Omega; \Phi \vdash H$, $\Delta_1; \Omega, \Phi \vdash \mathbb{E}[e] : \tau; \Delta_2$ *and* $\Omega; H; \mathbb{E}[e] \longrightarrow \Omega; H'; \mathbb{E}[e']$, *then there exists* $\Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1$ *and* $\Delta_2' \supseteq \Delta_2$ *such that*

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

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

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

❑

B.1.20 Lemma (Preservation). *If* $\emptyset \vdash \Omega; H; e : \tau$ *then*

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

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

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

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

➤*Case* $\Omega; H; e \xrightarrow{\text{upd}} \Omega'; H'; e'$ **:** This transition must be by the update rule, therefore $e = \mathbb{E}[\mathbf{update}^{\Delta}]$ for some $\mathbb{E}$ and Δ; and either

(a) $\text{updateOK}(\text{upd}, \Omega, H, \Delta) \quad \Omega' = \mathcal{U}[\Omega]^{\text{upd}} \quad H' = \mathcal{U}[H]^{\text{upd}} \quad e' = \mathcal{U}[\mathbb{E}[0]]^{\text{upd}}$

(b) $e' = \mathbb{E}[1]$

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

B.1.21 Lemma (Progress). *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Delta_1; \Omega, \Phi \vdash_{\mu} e : \tau; \Delta_2$ *then either*

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

(ii) e *is a value*

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

➤*Case* (A.EXPR.INT | XVAR | LOC) **:** All values.

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

➤*Case* (A.EXPR.APP) **:** Assume

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

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

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

By IH one of (i)-(iii) holds for e_1:

➤*Case* (i) holds for e_1 **:** (i) holds for $e_1\ e_2$ by cong rule.

➤*Case* (iii) holds for e_1 **:** By IH there are three cases to consider for e_1:

➤*Case* (i) holds for e_2 **:** (i) holds for $e_1\ e_2$ by cong rule

➤*Case* (iii) holds for e_2 **:**

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

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

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

➤*Case* (A.EXPR.ABS) **:** Assume

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

By IH there are three cases to consider:

➤*Case* (i) holds for e **:** By cong reduction rule (i) holds.

➤*Case* (iii) holds for e **:** e is a value by case split, thus $\mathbf{abs}_t$ is also a value (by inspection of values)

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

The rest of the cases are similar.

❑

B.1.22 Theorem (Type Soundness). *If $\emptyset \vdash_\mu \Omega; H; e : \tau$ then either*

(i) there exists Ω', H', e' such that $\Omega; ; He \longrightarrow \Omega'; H'; e'$ and $\emptyset \vdash_\mu \Omega'; H'; e' : \tau$ or

(ii) e is a value

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

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

(b) e is a value

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

Bibliography

[Abr90] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Proc. 17th POPL*, pages 31–46, 1990.

[AFM$^+$95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proc. 22nd POPL*, pages 233–246, 1995.

[App94] Andrew Appel. Hot-Sliding in ML, December 1994. Unpublished manuscript.

[AV91] J. L. Armstrong and R. Virding. Erlang — An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., 1996.

[BAS$^+$05] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, and Robert W. Wisniewski. Providing dynamic update in an operating system. In *Proc. USENIX Annual Technical Conference*, 2005. To appear.

[BE03] M. Barr and S. Eisenbach. Safe Upgrading without Restarting. In *IEEE Conference on Software Maintenance (ICSM 2003)*. IEEE, Sept 2003.

[BH00] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[BHS$^+$03a] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update with destruct-time λ. In *Proc. ACM International Conference on Functional Programming (ICFP)*, August 2003.

[BHS$^+$03b] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. Technical Report UCAM-CL-TR-568, University of Cambridge Computer Laboratory, June 2003.

[BHSS03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of USE 2003: the Second International Workshop on Unanticipated Software Evolution (Warsaw)*, April 2003.

[BLS$^+$03] C. Boyapati, B. Liskov, L. Shrira, C-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[BMP$^+$04] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *SIGPLAN Not.*, 39(11):235–247, 2004.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[CEM04] Robert Chatley, Susan Eisenbach, and Jeff Magee. Magicbeans: a Platform for Deploying Plugin Components. In *Proceedings of Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.

[dB72] N. de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–391, 1972.

[dB80] N. de Bruijn. A survey of the project AUTOMATH. *To H. B. Curry: essays on combinatory logic, lambda calculus, and formalism*, pages 597–606, 1980.

[DE02] S. Drossopoulou and S. Eisenbach. Manifestations of Dynamic Linking. In *The First Workshop on Unanticipated Software Evolution (USE 2002)*, Málaga, Spain, June 2002. http://joint.org/use2002/proceedings.html.

[DE03] S. Drossopoulou and S. Eisenbach. Flexible, source level dynamic linking and re-linking. In *Proc. ECOOP 2003 Workshop on Formal Techniques for Java Programs*, 2003.

[DEW99] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a Model of Separate Compilation, Linking and Binary Compatibility. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 147–156. IEEE, July 1999.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[Dmi01a] Mikhail Dmitriev. Safe class and data evolution in large and long-lived java applications. Technical report, University of Glasgow, March 2001.

[Dmi01b] Mikhail Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution*, October 2001.

[Dug01] D. Duggan. Type-based hot swapping of running modules. In *Proc. ACM International Conference on Functional Programming (ICFP)*, 2001.

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java Binary Compatibility? In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 341–361. ACM Press, 1998.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[FF87] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–219. Elsevier North-Holland, 1987.

[FS91] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, September 1991.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[GKW97] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, The University of Edinburgh, December 1997.

[Gup94] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[Hei92] N. Heintze. *Set-Based Program Analysis*. PhD thesis, Department of Computer Science, Carnegie Mellon University, October 1992.

[HG98] G. Hjálmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX Annual Technical Conference*, June 1998.

[Hic01] M. W. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, August 2001.

[Hir03] Tom Hirschowitz. *Modules mixins, modules et récursion étendue en appel par valeur*. Thèse de doctorat, Université Paris 7, 2003.

[HLW03] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

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

[L$^+$01] X. Leroy et al. The Objective Caml system release 3.04 documentation, December 2001.

[Lee83] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, April 1983.

[LSS04] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. ACM Conference on Architectural support for programming languages and operating systems*, pages 211–223. ACM Press, 2004.

[MCG$^+$99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPG$^+$00] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2000.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 72–83, June 2006.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[ORH02] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of Java software. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2002.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[PHL97] John Peterson, Paul Hudak, and Gary Shu Ling. Principled dynamic code improvement. Technical Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pla97] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, 1997.

[SAH$^+$03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System Support for Online Reconfiguration. In *Proc. USENIX Annual Technical Conference*, June 2003.

[SC05] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2005.

[SHB$^+$05] 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 sysposium on Principles of programming languages*, pages 183–194, New York, NY, USA, 2005. ACM Press.

[Smi88] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review, SIGOPS*, 22(3):28–40, 1988.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Wal00] D. Walker. A type system for expressive security policies. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 254–267, January 2000.

[WCM00] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WN79] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, 1979.