UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Behavioural Properties and Dynamic Software Update for Concurrent Programs

by

Gabrielle Anderson

Thesis for the degree of Doctor of Philosophy

April 2013

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

BEHAVIOURAL PROPERTIES AND DYNAMIC SOFTWARE UPDATE FOR CONCURRENT PROGRAMS

by Gabrielle Anderson

Software maintenance is a major part of the development cycle. The traditional methodology for rolling out an update to existing programs is to shut down the system, modify the binary, and restart the program. Downtime has significant disadvantages. In response to such concerns, researchers and practitioners have investigated how to perform update on running programs whilst maintaining various desired properties. In a multithreaded setting this is further complicated by the interleaving of different threads' actions. In this thesis we investigate how to prove that safety and liveness are preserved when updating a program. We present two possible approaches; the main intuition behind each of these is to find quiescent points where updates are safe. The first approach requires global synchronisation, and is more generally applicable, but can delay updates indefinitely. The second restricts the class of programs that can be updated, but permits update without global synchronisation, and guarantees application of update. We provide full proofs of all relevant properties.

Contents

List of Figures

Declaration of Authorship

I, Gabrielle Anderson , declare that the thesis entitled *Behavioural Properties and Dynamic Software Update for Concurrent Programs* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: (Anderson and Rathke, 2009, 2011; Anderson, 2011; Anderson and Rathke, 2012)

Signed:...

Date:..

Acknowledgements

The insight that the synchronisation behaviour inherent in blocking message passing programs could be used for co-ordinating updates came fairly early in the course of my doctoral work. One of the most challenging aspects of the work, although being preliminary to the problem of DSU for message passing programs, was how to formalise the synchronisation behaviour in static programs. Existing work on Session Typing provided a general description of the synchronisation behaviour. The way that the available formalisations relied on the synchronisation behaviour, however, was unclear. In order to use the behaviour in the more complex DSU setting, I had to unravel the details of the existing work, and create a new formalisation. One thing that I have learned is that, when building on the work of others, one must either work with the original authors, or budget a lot of time for the preliminary comprehension stage.

For most of the duration of my doctoral work the only people working on problems to do with type systems and static analyses in my university were my supervisor and myself. This meant that I had few opportunities, besides those with Julian, to bounce ideas around. On the occasions that I had the opportunity to attend conferences, workshops, and summer schools, I found the academic discussions that I had to be hugely fulfilling. I very much enjoyed hearing about how other people used such techniques, about the problems they were solving, and about the new ideas they had that I could include in my work. I now am very much of the opinion that having a thriving community in order to cross-pollinate ideas to be an essential part of the research process.

As this period of my life comes to a close, I am grateful for the lessons I've learned, the mistakes I've made, the people who've helped me through them, and the new opportunities on the horizon.

Chapter 1

Introduction

Software maintenance is a major part of the development cycle; at one point it was estimated that "60 percent of all business expenditures on computing are for maintenance of software written in Cobol" (Freeman, 1986; Banker et al., 1993). Another study estimated that 67% of life-cycle costs are in the operations and maintenance phase (Arthur, 1988). The traditional methodology for rolling out an update to existing programs is to shut down the system, modify the binary, and restart the program. Halting a program to apply updates is, however, problematic. Downtime can have significant financial costs, often running into the millions of dollars (Scott, 1998). In addition, in some real-time and critical systems, downtime would break key safety properties, and would prevent updates from occurring at all. As a result, dynamic update of running code is the third most requested enhancement for Java (Sun-Microsystems, 2006). In response to such concerns, researchers and practitioners have investigated performing update on programs while they are still running.

Performing arbitrary updates to programs, at unspecified points during their execution, can lead to arbitrary program behaviour; this negates our attempt to maintain the program's service and availability while performing an update. In order to maintain well formed programs it is therefore essential to place some restrictions on what updates can occur, and when they can occur. Dynamic Software Update (DSU) comprises a series of approaches that update running programs, while providing a degree of safety, flexibility, and efficiency (Subramanian et al., 2009).

Ensuring good behaviour of concurrent (multi-threaded) programs is notoriously difficult (Lea, 1999). When multiple threads can interleave in each other's execution, a variety of interesting errors can occur. Data races consist of several processes accessing some shared state in such a way that each can see, and interfere with, intermediary values in each others' state. Communication errors occur when values are sent over message queues and are subsequently used under incorrect typing assumptions, for example using an integer as an object reference. In addition, interleaving introduces a new potential problem for concurrent programs: liveness. Deadlock occurs when a program can never proceed, and livelock occurs when part of a program continually executes, performing no useful computation. Concurrent programs often include programming constructs which can either reduce or block, depending on some accompanying state, such as those for locking mutexes or receiving data. In many cases this blocking behaviour is used to eliminate data races or communication errors. There is often a tension between providing safety guarantees and providing guarantees of deadlock absence (Lea, 1999), especially for mutual exclusion; in a program with more blocking it is easier to prove safety, but harder to prove liveness, and vice versa.

The majority of existing work on Dynamic Software Update concerns single-threaded programs (Gupta et al., 1996; Bierman et al., 2003; Boyapati et al., 2003; Stoyle et al., 2005; Neamtiu et al., 2008; Bierman et al., 2008; Kaashoek and Arnold, 2009). These have focused primarily on preserving type safety, both with respect to changing function signatures (Bierman et al., 2003; Stoyle et al., 2005) and changing class signatures (Boyapati et al., 2003). Whilst type safety is important in updates for concurrent programs, and indeed has been thoroughly considered (Neamtiu and Hicks, 2009; Subramanian et al., 2009), the concern of many concurrent program designers is focused elsewhere on behavioural properties such as liveness. In addition, the techniques developed for updating single-threaded programs rely on finding "safe" points during the execution, and applying the update at this point. The extension of this methodology to multi-threaded programs requires waiting until all threads happen to be at a safe point at the same time, or forcing threads to block when they reach a safe point, until all threads are globally synchronised. The result of this approach is delay during synchronisation, at best, and deadlock at worst, and can delay the application of updates indefinitely (Subramanian et al., 2009).

In this thesis we investigate how to prove that safety and liveness are preserved when updating a program. To this end we first consider how to prove these properties for programs in the absence of update (Chapter 3). We focus on programs that use message passing for shared state and synchronisation. We then consider how to update such programs while preserving safety and liveness (Chapter 4). We consider two specific approaches for update. The first is more straightforward and simpler to reason about, but requires global synchronisation. The second performs update using a less restrictive form of synchronisation, with which we can update threads separately; unfortunately this approach is significantly more complex to reason about, and is applicable to fewer programs than the first approach. Finally, we discuss our intuition on how to show that an update will not be delayed indefinitely.

1.1 Safety and Liveness of Concurrent Programs

Safety and liveness are two key issues for concurrent programs (Lea, 1999). Safety denotes that nothing bad ever happens. Liveness denotes that something good eventually happens. Safety is important for both single-threaded and multi-threaded programs, as many of the same errors can occur. The addition of concurrency, however, introduces a wide variety of issues for safety, and the issue of liveness.

Message passing is an idiom for concurrent programs, where processes use messages to communicate and synchronise. This obviates the need for direct shared state synchronisation. Session Typing is a formalism used to structure interaction between message passing processes, ranging from low level message passing programs to web orchestration and high level distributed web systems. By abstracting the details of such systems away to the core communication behaviour it is possible to prove a wide variety of properties. These properties include deadlock freedom and absence of communication errors (Honda, 1993; Honda et al., 1998; Bettini et al., 2008), safe re-orderings of communication actions (Gay and Hole, 2005; Mostrous et al., 2009; Deniélou and Yoshida, 2010), and bounds on channel buffer sizes (Deniélou and Yoshida, 2010; Gay and Vasconcelos, 2010).

The proofs of properties for message passing programs rely heavily on the synchronisation semantics of the communication actions; a program that does not block when it attempts to receive from an empty message queue has fewer places where it can deadlock, but the increased flexibility and decreased synchronisation makes communication safety more difficult to prove. Existing session typing work focuses on both synchronous and asynchronous communication, with blocking receives and non-blocking sends. The dependencies in the proofs on the communication semantics is often unclear and left implicit. This makes it difficult to re-factor results to message passing languages with even minor differences in communication semantics.

We consider how to prove communication safety and liveness in a general functional language with accompanying state. The general methodology amounts to model checking, and as such is computationally expensive. We then proceed to consider stronger properties that imply general safety, as these stronger properties are often less expensive to prove. We re-prove existing results of safety and liveness for message passing programs that use blocking receive semantics, using a novel and intuitive induction. We then prove safety and liveness for message passing programs that use a non-blocking receive semantics, using a similar induction. We argue that the methodology of a general property, and proving stronger inductive properties that imply it, aids the comprehension of the proofs, and aids the process of writing new proofs when small changes occur to the message passing semantics.

1.2 Dynamic Software Update

Safety and liveness are two key issues for concurrent programs. Hence when we update concurrent programs we should consider whether an update violates safety or liveness properties. We present a framework for updating programs written in a functional language with accompanying state based on that presented in Chapter 3. We show how to extend the general safety and liveness properties to accommodate updates. We then show how to prove safety for particular approaches of updates to message passing programs.

Update Framework: Updates can be introduced into a running program at any time. This reflects the fact that updates are written and rolled out at some arbitrary point after a program has begun execution. Updates are not necessarily applied as soon as they are introduced - in order to prove safety and liveness we must restrict when updates can occur. An update is only applied after some property of the code and the state becomes true, for example reaching a particular point in the code. An update replaces annotated regions of the code with new bodies of code.

Global Typability Approach: This approach consists of a form of global synchronisation. We assume that a program is following an iterative protocol, performing the same communication actions in a loop. We show that if all the threads are in the same iteration of the protocol, it is safe to perform the update. This approach, while requiring global synchronisation, is straightforward to understand and prove safe. We can consider the protocol being used and show how, in certain cases, the window when all threads are updatable occurs infinitely often, and hence an update will not be delayed indefinitely.

Local Update Approach: This approach is significantly more complex than the Global approach. We again assume that a program is following an iterative protocol, performing the same communication actions in a loop. The intuition of this approach is as follows. The thread that is at the largest iteration number is updated first. The remaining threads are updated when they reach the iteration number when the lead thread(s) updated. This approach ensures that all messages used under the old protocol are used up. Interestingly, we can perform this approach without actually keeping track of the iteration numbers of the individual threads; this information can be inferred from the code and the state. Using this approach we can update threads separately, without need for global synchronisation or reference execution point of other threads.

Chapter 2

Background Material and Related Work

In this chapter we present an overview of background material relevant to this thesis, and its context in the literature. In Section 2.1 we explore Type and Effect Systems: these are static analyses that can be used to abstract the side effecting (*impure*) behaviour of a program such as network or file accesses. In Section 2.2 we describe Session Typing analyses: these place constraints on the impure behaviour of message passing programs that are sufficient to prove various useful properties of concurrent programs such as liveness. In Section 2.3 we explore Dynamic Software Update. We describe common design decisions, with a particular emphasis on how these decisions influence interesting properties such as type safety.

2.1 Type and Effect Systems

An example type and effect analysis is one that determines the reference variables that are used in a program, and how they are used (Figure 2.3), as in (Nielson and Nielson, 1999). This information could be useful for static deallocation or for security to guarantee that a program only performs read actions on certain variables. We consider a simple call by value lambda calculus with reference variables, along the lines of ML (Figure 2.1). The language consists of values, functions, function applications, and reference variable constructs. The construct $\texttt{new}_\pi\, x \;:=\; t_1 \,\texttt{in}\, t_2$ creates a new reference variable x with an annotation π; it evaluates t_1 to a value and binds it as x within t_2. The

$$t \;::=\; v \mid \lambda x.t \mid t\,t \mid \texttt{new}_\pi\, x \;:=\; t\,\texttt{in}\,t \mid\, !x \mid x := t \qquad\qquad v \;::=\; n \mid b$$

Figure 2.1: Lambda Calculus

$$T \;::=\; \textsc{Int} \mid \textsc{Bool} \mid T\,\texttt{ref}\,\pi \mid T \xrightarrow{\varphi} T \qquad \varphi \;::=\; \{!\pi\} \mid \{\pi :=\} \mid \{\texttt{new}\,\pi\} \mid \varphi \cup \varphi \mid \emptyset$$

Figure 2.2: Types and Effects

$$\frac{}{\emptyset \wr \Gamma \vdash n\colon \textsc{Int}} \qquad \frac{}{\emptyset \wr \Gamma \vdash b\colon \textsc{Bool}} \qquad \frac{}{\emptyset \wr \Gamma \vdash x\colon \Gamma(x)}$$

$$\frac{\varphi \wr \Gamma, x\colon T_1 \vdash t\colon T_2}{\emptyset \wr \Gamma \vdash \lambda x.t\colon T_1 \xrightarrow{\varphi} T_2} \qquad \frac{\varphi_1 \wr \Gamma \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma \vdash t_2\colon T_2}{\varphi_1 \cup \varphi_2 \cup \varphi_3 \wr \Gamma \vdash t_1\, t_2\colon T_1}$$

$$\frac{\Gamma(x) = T\,\texttt{ref}\,\pi}{\{!\pi\} \wr \Gamma \vdash !x\colon T} \qquad \frac{\varphi \wr \Gamma \vdash t\colon T \qquad \Gamma(x) = T\,\texttt{ref}\,\pi}{\varphi \cup \{\pi :=\} \wr \Gamma \vdash x := t\colon T}$$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1\colon T_1 \qquad \varphi_2 \wr \Gamma, x\colon T_1\,\texttt{ref}\,\pi \vdash t_2\colon T_2}{\varphi_1 \cup \varphi_2 \cup \{\texttt{new}\,\pi\} \wr \Gamma \vdash \texttt{new}_\pi\, x \;:=\; t_1\,\texttt{in}\,t_2\colon T_2}$$

Figure 2.3: Side Effect Analysis

annotation π is unique, and is used to differentiate accesses to this reference variable from accesses to other reference variables. The construct $!x$ dereferences the variable x, and $x := t$ assigns to x the value obtained by reducing t to a value. This language has a conventional call by value semantics, and makes use of standard syntactic sugaring such as $t_1;\, t_2$ for $(\lambda x.t_2)\, t_1$ where x is not free in t_2

In order to describe the side effects of programs we make use of types and effects (Figure 2.2). The effect $\{!\pi_1\}$ denotes the dereference of the variable that has annotation π_1. The effect $\{\pi_2 :=\}$ denotes assignment to the variable that has annotation π_1. The effect $\{\texttt{new}\,\pi_3\}$ denotes the creation of a new variable that has a new unique annotation π_3. The primitive types are straightforward. The reference variable type $T\,\texttt{ref}\,\pi$ denotes a reference variable that contains a value of type T and has annotation π. The function type includes an effect annotation φ on its arrow, referred to as the *latent effect*; this denotes the side effect that will occur by when the function is applied to an argument. The effect of the function does not appear in the effect of a program until the function is applied, because the body of the function (from which the effect φ comes) is not evaluated until that point.

We make use of the judgement $\varphi \wr \Gamma \vdash t\colon T$ which denotes that, under type variable assumptions Γ, the program t will reduce to a value of type T, and its side effect is described by the effect φ. The effect φ is generally only a conservative approximation of the effect of a program due to undecidability of determining the actual effect (for example, we do not know which path will be taken at a conditional without running the program, which itself is undecidable). We specify the side effect analysis in Figure 2.3. Values and variables have no side effects. Functions, taken alone, also have no effects:

the effect of the function body is annotated on the type of the function. When a function is applied, the effect of the application construct is the effect φ_1 of reducing the left hand side to a function, the effect φ_2 of reducing the right hand side to a value, and the effect φ_3 of reducing the function body, where φ_3 is the latent effect annotated on the type of the function. The dereference of a variable x has the effect $\{!\pi\}$ where π is the annotation associated with x. Assignment of t to a variable x has the effect $\varphi \cup \{\pi :=\}$, where π is the annotation associated with the variable, and φ is the side effect of reducing t to the value which will be assigned to the variable. The creation a new reference variable has the effect $\varphi_1 \cup \varphi_2 \cup \{\texttt{new}\,\pi\}$, where φ_1 is the effect of reducing t_1 to the value assigned to x, t_2 is the effect of the body in which x is bound, and $\{\texttt{new}\,\pi\}$ is the effect of creating a reference variable with annotation π.

An example program that we can analyse is the following:

$$(\lambda y.!y)\,(\texttt{new}_\pi\, x\ :=\ 1\,\texttt{in}\,x) \tag{2.1}$$

This program creates a new reference variable annotated by π, assigns the value 1 to it, and passes it to a function which dereferences it. We can use the typing rules in Figure 2.3 to generate the possible side effect behaviour of the program:

$$\dfrac{\dfrac{\dfrac{(y\colon \textsc{Int}\,\texttt{ref}\,\pi)(y) = \textsc{Int}\,\texttt{ref}\,\pi}{\{!\pi\} \wr y\colon \textsc{Int}\,\texttt{ref}\,\pi \vdash !y\colon \textsc{Int}}}{\emptyset \wr \emptyset \vdash \lambda y.!y\colon \textsc{Int}\,\texttt{ref}\,\pi \xrightarrow{\{!\pi\}} \textsc{Int}} \qquad \dfrac{\emptyset \wr \emptyset \vdash 1\colon \textsc{Int} \qquad \dfrac{(x\colon \textsc{Int}\,\texttt{ref}\,\pi)(x) = \textsc{Int}\,\texttt{ref}\,\pi}{\emptyset \wr x\colon \textsc{Int}\,\texttt{ref}\,\pi \vdash x\colon \textsc{Int}\,\texttt{ref}\,\pi}}{\{\texttt{new}\,\pi\} \wr \Gamma \vdash \texttt{new}_\pi\, x\ :=\ 1\,\texttt{in}\,x\colon \textsc{Int}\,\texttt{ref}\,\pi}}{\{\texttt{new}\,\pi, !\pi\} \wr \emptyset \vdash (\lambda y.!y)\,(\texttt{new}_\pi\, x\ :=\ 1\,\texttt{in}\,x)\colon T} \tag{2.2}$$

The above program has the side effects of creating a reference variable with the annotation π, and dereferencing of a reference variable with the same annotation, leading to an overall effect of $\{\texttt{new}\,\pi, !\pi\}$.

This analysis exemplifies the key aspects of type and effect systems. Certain primitives and constructs within the language are annotated with labels, and the remaining primitives have no effect. The effects of non-annotated constructs are constructed by combining the effects of the sub-components. Functions have their effect annotated on their type, and this effect is ignored until the function is applied to an argument.

In this thesis we use type and effect systems to abstract from a source language to an effect representation which is similar to a process calculus. The effect representation denotes both the communication actions that occur, and their temporal ordering. We do not make any contributions to the type and effect literature; we use it as a tool.

$$
\begin{array}{lcll}
t & ::= & & \textit{Terms} \\
 & | & e & \text{Simple expressions} \\
 & | & \texttt{snd}(c, e).t & \text{Send} \\
 & | & \texttt{rcv}(c)(x : T).t & \text{Receive} \\
 & | & \texttt{if}\, e\, \texttt{then}\, t\, \texttt{else}\, t & \text{Conditional} \\
 & | & \texttt{sel}(c, l).t & \text{Selection} \\
 & | & \texttt{case}\ c\ \texttt{of}\ \{\widetilde{l \mapsto t}\} & \text{Branching} \\
 & | & \mu \underline{X}.t & \text{Recursion} \\
 & | & \underline{X} & \text{Recursion variable} \\
 & | & \texttt{error} & \text{Error} \\
 & | & \texttt{0} & \text{End} \\
 \\
e & ::= & e + e, e \wedge e, \ldots & \textit{Simple expressions} \\
 \\
P & ::= & & \textit{Processes} \\
 & | & t & \text{Single process} \\
 & | & P \parallel P & \text{Parallel processes}
\end{array}
\qquad
\begin{array}{lcll}
v & ::= & & \textit{Values} \\
 & | & n & \text{Integers} \\
 & | & b & \text{Booleans} \\
 & | & () & \text{Unit} \\
 \\
\sigma & ::= & & \textit{Named channel queues} \\
 & | & c \mapsto q & \text{Map from channel name to queue} \\
 & | & \sigma, \sigma & \text{Set of named channel queues} \\
 & | & \emptyset & \text{Empty set of queues} \\
 \\
q & ::= & & \textit{Queues} \\
 & | & v & \text{Value} \\
 & | & l & \text{Label} \\
 & | & q, q & \text{Queue composition} \\
 & | & \emptyset & \text{Empty queue}
\end{array}
$$

Figure 2.4: Message Passing Language and Resource Syntaxes

Type and effect systems are first proposed in (Gifford and Lucassen, 1986) in order to statically track the allocation, access, and deallocation of dynamically allocated memory. This work is expanded and simplified in (Nielson and Nielson, 1999) which shows how type and effect systems can also be used for information control in security, procedure call site analysis, side effect analysis, and simple message passing analysis. Type and effect systems are used many settings, including enforcing locking disciplines in Java (Flanagan and Freund, 2000), in policy-based access control (Bartoletti et al., 2007), and Dynamic Software Update (Neamtiu et al., 2008). A more detailed overview can be seen in (Marino and Millstein, 2008).

2.2 Session Typing

Session Typing analyses consist of a type and effect analysis, which obtains the communications behaviour of a program, and various constraints placed on what effects are accepted by the analysis. A key notion in Session Typing is duality, where certain effects fit together if the types of the values and labels which are sent and received are consistent. Systems with certain duality or complementarity properties can be shown to have certain safety and liveness properties, including communication safety, session fidelity, and deadlock freedom.

$$\frac{[\![e]\!] = v \qquad \sigma' = \sigma[c \mapsto q, v]}{[\sigma[c \mapsto q]]\ \texttt{snd}(c, e).t \to [\sigma']\ t} \qquad \frac{v : T \qquad \sigma' = \sigma[c \mapsto q]}{[\sigma[c \mapsto v, q]]\ \texttt{rcv}(c)(x : T).t \to [\sigma']\ t[v/x]}$$

$$\frac{v : T' \neq T \qquad \sigma' = \sigma[c \mapsto q]}{[\sigma[c \mapsto v, q]]\ \texttt{rcv}(c)(x : T).t \to [\sigma']\ \texttt{error}} \qquad \frac{e : T \qquad T \neq \textsc{Bool}}{[\sigma]\ \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to [\sigma]\ \texttt{error}}$$

$$\frac{e \downarrow \texttt{true}}{[\sigma]\ \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to [\sigma]\ t_2} \qquad \frac{e \downarrow \texttt{true}}{[\sigma]\ \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to [\sigma]\ t_3}$$

$$\frac{}{[\sigma[c \mapsto q]]\ \texttt{sel}(c, l).t \to [\sigma[c \mapsto q, l]]\ t} \qquad \frac{l_i \in \widetilde{l} \qquad \sigma' = \sigma[c \mapsto q]}{[\sigma[c \mapsto l_i, q]]\ \texttt{case}\ c\ \texttt{of}\ \{\widetilde{l \mapsto t}\} \to [\sigma']\ t_i}$$

$$\frac{l \notin \widetilde{l}}{[\sigma[c \mapsto l, q]]\ \texttt{case}\ c\ \texttt{of}\ \{\widetilde{l \mapsto t}\} \to [\sigma[c \mapsto l, q]]\ \texttt{error}}$$

$$\frac{}{[\sigma]\ \mu \underline{X}.t \to [\sigma]\ t[\mu \underline{X}.t/\underline{X}]} \qquad \frac{}{[\sigma]\ f(t) \to [\sigma]\ t}$$

$$\frac{[\sigma]\ P_1 \to [\sigma']\ P_1'}{[\sigma]\ P_1 \,||\, P_2 \to [\sigma']\ P_1' \,||\, P_2} \qquad \frac{[\sigma]\ P_2 \to [\sigma']\ P_2'}{[\sigma]\ P_1 \,||\, P_2 \to [\sigma']\ P_1 \,||\, P_2'}$$

Figure 2.5: Message Passing Semantics

$$\varphi \ ::= \ c!\langle T \rangle;\, \varphi \mid c?(T);\, \varphi \mid \varphi \oplus \varphi \mid c\&\{\widetilde{l \mapsto \varphi}\} \mid \mu \underline{X}.\varphi \mid \underline{X} \mid 0$$

$$\Phi \ ::= \ \varphi \mid \Phi \parallel \Phi \qquad T \ ::= \ \textsc{Int} \mid \textsc{Bool} \mid \textsc{Unit}$$

Figure 2.6: Types and Effects for Message Passing Language

In order to discuss Session Typing we present a simple message passing system. We define the language in Figure 2.4. The transmission of values is performed by $\texttt{snd}(c, e).t$, which reduces e to a value, adds that value onto the end of the queue for channel c, and continues with t. The reception of values is performed by $\texttt{rcv}(c)(x : T).t$, which takes the value off the front of the message queue for c, and substitutes it for x in t. Conditional statements $\texttt{if}\,e\,\texttt{then}\,t\,\texttt{else}\,t$ reduce the pure (non side-effecting) guard, and then branches depending on the value. Selection statements $\texttt{sel}(c, l).t$ is used to signal a choice in protocol direction, in particular, which option a branch (that exists in another term with which the selecting term is interacting) should take. These put the relevant label on the end of the message queue for c. Branching statements $\texttt{case}\ c\ \texttt{of}\ \{\widetilde{l \mapsto t}\}$ provides a series of options between which another term can choose. These take the value off the front of the message queue for c, and branch over the value received. Recursion, errors, and blocked terms are standard. We specify the semantics formally in Figure 2.5.

Choice primitives can be used to specify more complex communication patterns, such as the following ATM example (Carbone et al., 2007):

$$
\begin{array}{l}
c_1!\langle \texttt{deposit}\rangle.c_1!\langle \texttt{amount}\rangle.c_1!\langle 300\rangle.c_2!\langle \texttt{balance} \mapsto c_2?(x).()\rangle.\ \| \\
c_1\&\{\texttt{deposit} \mapsto (c_1\&\{\texttt{amount} \mapsto c_1?(x).c_2!\langle \texttt{balance}\rangle.c_2!\langle(50+x)\rangle.()\}), \\
\quad \texttt{withdraw} \mapsto c_1\&\{\texttt{amount} \mapsto c_1?(y).\texttt{if}\ (x \leq 50)\ \texttt{then}\ c_2!\langle \texttt{dispense}\rangle.() \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ c_2!\langle \texttt{overdraft}\rangle.() \\
\qquad\qquad\qquad\qquad \} \\
\quad \}
\end{array}
\tag{2.3}
$$

This example consists of two participants, a user (the first thread) and the ATM (the second thread). The ATM presents two services to the user: depositing money into the account, which returns the balance after the deposit, and withdrawing money using the ATM, both assume there is £50 in the account before any action. In this example the user chooses to deposit £300 into the account. The offering of different services, and the use of services to annotate the semantic meaning of values being sent back and forth, demonstrates the complexity of communications behaviour which can be represented using Session Types. Many Session Typing systems also include the delegation communication primitive; this permits channels (and the responsibilities entailed by the Session Typing analysis, below) to be sent over channels. For simplicity we do not consider delegation.

We represent the communication behaviour of a program by effects (which are referred to as Local Session Types) (Honda, 1993; Mostrous et al., 2009; Bettini et al., 2008; Yoshida and Vasconcelos, 2006; Honda et al., 2008; Gay et al., 2003; Gay and Vasconcelos, 2007). These are defined formally in Figure 2.6. The structure follows that of the message passing terms. Send and receive actions describe the transmission and reception of a value of type T on channel c, followed by the continuation φ. Non-deterministic choice between possible effects φ_1 and φ_2 is written as $\varphi_1 \oplus \varphi_2$. External choice, based on another participant's choice, is written as $c\&\{\widetilde{l \mapsto t}\}$. Recursion, errors, and blocked terms are standard. We define a simple type and effect analysis for message passing systems in Figure 2.7 to abstract a program's communication behaviour.

Communication Errors occur when a thread receives and uses a value which is of a type that it doesn't expect. An example reduction sequence with an error is:

$$
\begin{array}{ll}
1 & \boxed{[c \mapsto \emptyset]}\ \texttt{snd}(c, \texttt{true}).0 \parallel \texttt{rcv}(c)(x : \textsc{Int}).x + 4 \\
2 & \quad \to \boxed{[c \mapsto \texttt{true}]}\ 0 \parallel \texttt{rcv}(c)(x : \textsc{Int}).x + 4 \\
3 & \quad \to \boxed{[c \mapsto \emptyset]}\ 0 \parallel \texttt{true} + 4 \\
4 & \quad \to \boxed{[c \mapsto \emptyset]}\ 0 \parallel \texttt{error}
\end{array}
\tag{2.4}
$$

$$\frac{}{0 \wr \Gamma \vdash n \colon \textsc{Int}} \qquad \frac{}{0 \wr \Gamma \vdash b \colon \textsc{Bool}} \qquad \frac{}{0 \wr \Gamma \vdash () \colon \textsc{Unit}} \qquad \frac{}{0 \wr \Gamma \vdash x \colon \Gamma(x)}$$

$$\frac{0 \wr \Gamma \vdash e : T_1 \qquad \varphi \wr \Gamma \vdash t : T}{c!\langle T_1\rangle; \varphi \wr \Gamma \vdash \texttt{snd}(c, e).t : T} \qquad \frac{\varphi \wr \Gamma, x : T_1 \vdash t : T}{c?(T_1); \varphi \wr \Gamma \vdash \texttt{rcv}(c)(x : T).t : T}$$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T}{\varphi_1 \oplus \varphi_2 \wr \Gamma \vdash \texttt{if}\, e\, \texttt{then}\, t_1\, \texttt{else}\, t_2 \colon T} \qquad \frac{\varphi_i \wr \Gamma \vdash t_i : T}{c!\langle l\rangle; \varphi \vdash \texttt{sel}(c, l).t : T}$$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T}{c\&\{\widetilde{l \mapsto \varphi}\} \wr \Gamma \vdash c\&\{\widetilde{l \mapsto t}\} \colon T} \qquad \frac{\varphi \wr \Gamma \vdash t \colon T}{\mu \underline{X}.\varphi \wr \Gamma \vdash \mu \underline{X}.t \colon T}$$

$$\frac{}{\underline{X} \wr \Gamma \vdash \underline{X} \colon T} \qquad \frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi} \qquad \frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2}$$

Figure 2.7: Type and Effect Analysis for Message Passing Language

Session Typing analyses can be used to rule out this type of error. These analyses can be simplified if they only need consider the effect of a program, rather than the entire program itself. Hence Session Typing analyses require that the behaviour of a program is completely encompassed in that program's effect. In other words, whenever a program can perform a specific communication action, the effect of that program, as generated by the type and effect system, can perform an abstract version of the action. This property known as *fidelity* (Honda et al., 2008; Dezani-Ciancaglini and Liguoro, 2010). For example, an action $c!\langle e\rangle.$ is abstracted by $c!\langle T\rangle.$ where $e \colon T$.

The program in Example 2.4 can be represented using the effect:

$$c!\langle \textsc{Bool}\rangle; 0 \,||\, c?(\textsc{Bool}); 0 \tag{2.5}$$

This denotes that the program consists of two threads, one which will send a boolean value on a channel, and one which will receive a value from the same channel that it expects to be an integer.

The two threads' behaviour will obviously interact to cause an error, as the type being sent and the type expected to be received do not match up. We can formalise our intuition as to when a set of threads' behaviour fits together; when they do so their behaviour is called *complementary*. We formally define complementarity in Figure 2.8, where channels are assumed to be used in a uni-directional manner by two participants.

The Communication Safety property denotes that a thread will never receive a value which is of a different type to that which it is expecting. When a message passing system conforms to the above assumptions, and when all pairs of effects (from the two participants) on channels are complementary, it is possible to prove communication safety (Honda, 1993; Honda et al., 1998; Gay et al., 2003; Gay and Vasconcelos, 2007). A

$$\begin{array}{rcl} \mathsf{compl}(c!\langle T_1\rangle;\varphi_1, c?(T_1);\varphi_2) & \overset{\text{def}}{=} & T_1 = T_2 \wedge \mathsf{compl}(\varphi_1,\varphi_2) \\ \mathsf{compl}(\bigoplus_I c!\langle T_1\rangle;\varphi_i, c\&\{\widetilde{l_j \mapsto \phi'_j}\}_J) & \overset{\text{def}}{=} & \forall i \in I\, \exists j \in J.l_i = l_j \wedge \mathsf{compl}(\varphi_i,\varphi'_j) \\ \mathsf{compl}(0,0) & = & \texttt{true} \end{array}$$

Figure 2.8: Complementary Relation

basic Session Typing analysis can hence be seen as a type and effect analysis along with some restrictions on the permitted communication effects of message passing programs.

The concept of complementarity only fulfils our intuition for interactions which consist of two participants. Global Session Types generalise the notion of complementarity to describe how the interactions of an arbitrary number participants relate to each other (Honda et al., 2008). We define a simplified version of Global Session Types in Figure 2.9. The type $d_1 \to d_2 \colon c\langle T\rangle; G$ denotes that participant d_1 sends a value of type T over channel c to participant d_2, and then continues with the behaviour specified in G. This describes the duality of sends and receives expressed in complementarity. The type $d_1 \to d_2 \colon c\langle \widetilde{T \mapsto G}\rangle; G$ denotes that participant d_1 can make a choice (invoke a service) that is offered by participant d_2. This describes the duality of offering a service and service selection. The continuation for each construct may contain behaviour of participants other than d_1 and d_2, and hence we can represent temporal information about the global communications behaviour. Consider the following Global Session Type:

$$\begin{array}{l} d_1 \to d_2 \colon c\langle \textsc{Int}\rangle; \\ d_2 \to d_3 \colon c\langle \textsc{Bool}\rangle; \\ \mathsf{0} \end{array} \tag{2.6}$$

This denotes that participant d_2 will receive an integer from d_1, and then proceed to send a boolean to d_3. Using the global description of the communications behaviour it is possible to obtain a thread local description of the communications behaviour (Honda et al., 2008). We define the projection function $G \restriction d$, which generates the thread local behaviour described by a global protocol, in Figure 2.10. Global Session Types can be used to prove communication safety in systems with an arbitrary number of participants.

The earliest work on Session Typing focusses on two-party, synchronous communication (Honda, 1993). Honda considers how to rule out communication errors, how to rule out deadlock, and how to define bisimulation for Session Typing systems. Honda, Kubo, and Vasconcelos extend Honda's earlier work and describe how to represent method invocation, unbounded interaction patterns, and delegation using Session Typing (Honda et al., 1998). Some errors in (Honda, 1993; Honda et al., 1998) have been discovered and corrected (Gay and Hole, 2005; Yoshida and Vasconcelos, 2006).

$$G \ ::= \ d_1 \to d_2 \colon r\langle T\rangle;\, G \mid d_1 \to d_2 \colon r\langle \widetilde{T \mapsto G}\rangle \mid \mu\underline{X}.G \mid \underline{X} \mid \mathsf{0}$$

Figure 2.9: Global Session Types

$$\begin{array}{lcl}
d_1 \to d_2 \colon c\langle T\rangle;\, G' \restriction d_1 & \overset{\text{def}}{=} & c!\langle T\rangle.G' \restriction d_1 \\
d_1 \to d_2 \colon c\langle T\rangle;\, G' \restriction d_2 & \overset{\text{def}}{=} & c?(T).G' \restriction d_2 \\
d_1 \to d_2 \colon c\langle \widetilde{l \mapsto G}\rangle_I \restriction d_1 & \overset{\text{def}}{=} & \bigoplus_I c!\langle T_1\rangle; \varphi_i \\
d_1 \to d_2 \colon c\langle \widetilde{l \mapsto G}\rangle_I \restriction d_2 & \overset{\text{def}}{=} & c\&\{l \mapsto \widetilde{(G \restriction d_2)}\}_I \\
(\mu\underline{X}.G) \restriction d & \overset{\text{def}}{=} & \mu\underline{X}.(G \restriction d) \\
\underline{X} \restriction d & \overset{\text{def}}{=} & \underline{X} \\
\mathsf{0} \restriction d & \overset{\text{def}}{=} & \mathsf{0}
\end{array}$$

Figure 2.10: Global Session Type Projection Function

Whilst most work on Session Typing has made use of π calculus style calculi (Honda, 1993; Mostrous et al., 2009; Bettini et al., 2008; Yoshida and Vasconcelos, 2006; Honda et al., 2008), in some a λ calculus formulation is used (Gay et al., 2003; Gay and Vasconcelos, 2007, 2010). In (Gay et al., 2003) Gay, Vasconcelos, and Ravara provide subtyping for Session Types. In (Gay and Vasconcelos, 2007) Gay and Vasconcelos include a linear type system. Additionally, they crucially introduce Session Typing for asynchronous systems.

Honda, Yoshida and Carbone generalise Session Typing to sessions that can include more than two parties (multi-party Session Typing) in a π calculus setting (Honda et al., 2008). They introduce the concept of a Global Session Type, which represents communications protocols between more than two participants. The global view can be projected down to the local view of each role and how it should communicate with the other roles. The communications mechanisms are also expanded to include multicast sending. They extend work on delegation, where an entity can pass responsibility for performing certain actions within a session on to another entity. The formulation of delegation, however, is such that the delegation is not completely transparent; it exposes implementation details of the delegator to the delegate which could be either technically or commercially undesirable. Bettini, Coppo, Luca, and Dezani-Ciancaglini solve the above limitations and provide the first robust, multi-party, Session Typing system (Bettini et al., 2008). They also guarantee global progress, the property that well typed programs will not deadlock, and specifically that different global sessions will not interfere with and deadlock each other.

Subtyping for Session Types is introduced in (Gay and Hole, 2005) which permits Session Type specifications to be refined into a richer behaviour. The subtyping is extended in (Mostrous et al., 2009) to thread local asynchronous subtyping, which permits the send and receive actions performed by a thread to be re-ordered in certain circumstances. A simple example of this reordering is to permit moving sends in front of receives, which increases the asynchrony of the the interleaved behaviour whilst maintaining the safety and liveness properties. In (Deniélou and Yoshida, 2010; Gay and Vasconcelos, 2010), it is shown how buffers with unbounded size can be identified, and how buffer bounds can be extracted for bounded buffers, from the Global Session Type representation. It is also shown how to perform thread local asynchronous subtyping without changing buffer bounds.

A more detailed review of Session Typing literature can be found in (Dezani-Ciancaglini and Liguoro, 2010).

2.3 Dynamic Software Update

The principle aim of Dynamic Software Update is to reduce the disruption that normally occurs when programs need to be updated. When a program needs to be updated, the traditional approach is to schedule some down time for the service, to take down the program, update it, and restart it. Downtime can have financial costs (Scott, 1998) and also presents concerns in safety critical systems.

If a program or service is stateless, such as an http server, a service on redundant hardware can be used to handle requests whilst the primary service is updated (Stoyle, 2006). If state or interaction behaviour (such as a communications protocol) needs to be preserved over the update, however, using redundant hardware to maintain a service preserves the problem of new programs interacting with old state or protocols (Stoyle, 2006). Hence the primary concern of Dynamic Software Update is how to co-ordinate the old and new code interacting with the old and new data.

In this section we describe the key design decisions surrounding Dynamic Software Update, and consider the implications of said choices, with a particular emphasis on how they influence safety properties. We begin by detailing the design decisions concerning the semantics of a system that can be dynamically updated (Section 2.3.1). We then consider safety for DSU, and how the prior design decision influence what safety properties can be demonstrated (Section 2.3.2). We conclude with other important considerations, including ease of use, efficiency, and case studies (Section 2.3.3).

2.3.1 Update Design Decisions

The semantics of Dynamic Software Update can be described in several key areas. Updating code and data are intimately interlinked. Update linking is important to implement code and data updating. Finally, update timing considers when an update is applied, and how the structure of code can influence when it is practical to perform an update.

2.3.1.1 Updating Code

The most straightforward approach to updating code is to unload the entire binary of the old program and to load the entire binary of the new program. In order to structure and reason about updates code update is, for the most part, performed at a finer level of granularity, that of the function or method.

There are two principal approaches that can be taken when replacing a function (respectively method). The most common approach is to permit an update to replace the body of the function, but preserve (or refine/sub-type) the signature of the function (the types of the parameters, and the type of the return value) (Gilmore et al., 1998; Hjálmtÿsson and Gray, 1998; Orso et al., 2002; Bierman et al., 2008; Kaashoek and Arnold, 2009). Whilst this approach might seem restrictive, it is sufficient to cover the majority of modifications (Neamtiu et al., 2005; Subramanian, 2010). In particular it covers many security related updates (Kaashoek and Arnold, 2009); as delays to security updates until scheduled downtime leaves a large window of vulnerability, the ability to perform security updates without downtime is significant. The alternative is that an update can replace a function, and change the signature of the function, or even delete the function in its entirety. This approach is obviously more flexible, and is required for a significant minority of updates (Neamtiu et al., 2005; Subramanian, 2010).

In order to update functions, we need a way to access and manipulate them. This is primarily achieved using indirection (Subramanian et al., 2009; Hayden et al., 2011d, 2009; Neamtiu and Hicks, 2009; Boyapati et al., 2003; Sewell, 2001; Cook and Lee, 1983; Appel, 1994; Kaashoek and Arnold, 2009; Neamtiu et al., 2006; Baumann et al., 2007). Managed languages perform indirection as a matter of course, performing a table lookup on a function call to find the most relevant, or most recently compiled, version of a function. This functionality can be leveraged by rewriting the lookup table to point to the updated function body (Boyapati et al., 2003; Bierman et al., 2008; Subramanian et al., 2009). In non-managed languages such as C/C++, this indirection is often implemented on top of the program (Neamtiu et al., 2006), adding overhead during normal execution. An alternate approach is, when updating a function, to overwrite the first few assembly instructions of the function with a jump to the new definition; this approach has been implemented for the Linux kernel, and operates with almost no overhead (Kaashoek and Arnold, 2009).

It is also possible to update existing programs by directly manipulating functions that are on the stack (Makris and Bazzi, 2009). This approach uses annotation and special compilation to create threads that can be interrupted and updated at runtime. This approach relies on specialist knowledge from the programmer, who has to write a program that transforms the existing stack frame.

The code of a program need not be updated all at once. Indeed, most Dynamic Software Update approaches that use indirection modify the function definition, and next time that function is called the new version is used, but does not modify any instances of the function that are on the stack. This approach can result in old code and new code existing in the same program.

An alternative to replacing individual functions is to load an entirely new binary into memory, and to start at some predefined point in that program's execution (Hayden et al., 2011b,d). The ability to resume execution at points other than the start of the new program permits designers to, for example, skip initialising variables, which would erase existing data which we want to keep. Hayden et al. argue that the approach of updating individual functions requires complex tool support, is challenging to reason about, and can add overhead due to the indirection discussed above. Whether the complexity and overhead required in more complex analyses is justified is an open research topic.

2.3.1.2 Updating Data

Unless application data or interactions with other programs needs to be preserved over the update application, it is easier to update a service using redundant hardware. Hence programs where the old and new code interacts with the old and new data, and the co-ordination thereof, are the key concerns and research areas for Dynamic Software Update.

An updated program may use data with a different format to that of the previous version, for example a record could be changed from $\{\texttt{FirstName} : \mathit{String}, \texttt{LastName} : \mathit{String}\}$ to $\{\texttt{Name} : \mathit{String}, \texttt{Age} : \mathit{Integer}\}$ (Gurtner, 2011). In order for the new code to interact with the old data, it must be transformed into the new format before the new code accesses it. In addition we do not want the transformation to throw away the old data; a transformation that initialises all data to a default value is comparable to stopping the program and restarting, without performing DSU (Subramanian et al., 2009). One possible transformation function would take a record in the old format, create a record in the new format, concatenate the `FirstName` and `LastName` fields, put the in the `Name` field, place a default value in the `Age` field, and return the new record. In systems where the old and new code co-exist, and the data can be accessed by both, we similarly need a transformation function from the new to the old format, to be applied before the old code uses a new data value (Gurtner, 2011; Wernli et al., 2011). The state transformers can be automatically generated using initialisation of default values or copying (Subramanian et al., 2009), or written by the programmer when more complex transformations are required (Bierman et al., 2003; Stoyle et al., 2005; Soules et al., 2003; Hayden et al., 2011d).

2.3.1.3 Update Linking

Linking code to data at runtime is known as dynamic linking (Cardelli, 1997; Drossopoulou and Eisenbach, 2002). While the technology for linking identifiers to code and data exists, managing those identifiers, and co-ordinating when they can be updated, is not straightforward. Consider the following examples (Stoyle, 2006), where $\pi_1(x, x')$ denotes choosing the first item in a pair and $\pi_2(x, x')$ denotes choosing the second.

$$\texttt{let}\, x = (4,5)\,\texttt{in}\, \pi_1(\pi_1(x,x))x \tag{2.7}$$

Using a call by value reduction semantics this term could reduce as follows:

$$\texttt{let}\, x = (4,5)\,\texttt{in}\, \pi_1(\pi_1(x),x) \rightarrow \pi_1(\pi_1(4,5),(4,5)) \tag{2.8}$$

After this reduction it is no longer meaningful to discuss updating x. In order to maintain updatable identifiers as long as possible Stoyle shows how to delay substitution but maintain contextual equivalence (Stoyle, 2006). These techniques can be used to administrate re-linking at runtime and to increase the scope of where updates can be applied.

2.3.1.4 Update Timing

Updatable programs typically make use of specialised API calls to apply updates, possibly performing some dynamic checks beforehand to ensure the update is safe (Hayden et al., 2011d, 2009; Neamtiu and Hicks, 2009; Boyapati et al., 2003; Sewell, 2001; Kaashoek and Arnold, 2009; Neamtiu et al., 2006; Appel, 1994). Explicit language calls to an update API requires the programmer to have a working knowledge of update and requires them to reason about when would be a good time to perform updates, a significant addition in complexity when writing a program. In some cases the API calls can be automatically inserted (Stoyle et al., 2005).

One other option is to provide some external update manager that introduces the update without any knowledge of the original program (Baumann et al., 2007; Subramanian et al., 2009). The manager monitors the program to be updated, waiting for a suitable update points. One way to do this is to insert breaks (return barriers) at potential update points that return control to the monitor. Potential points include function entry and exit points, and loop back-edges (Subramanian et al., 2009).

When code is replaced using indirection, old code in existing stack frames is usually left unchanged. In such cases, irrespective of the point when the function bodies are replaced, the update doesn't really occur until the old code has returned.

In order for old code to continue to execute safely it must either not use any new data, or any new data must be transformed into an old format before use. The reverse is true for new code and old data. In order to simplify updates many systems employ *Representation Consistency* (Stoyle et al., 2005). This denotes that, after an update data is only in the new format. The conversion from old to new data can either be performed eagerly (Stoyle et al., 2005; Sewell, 2001; Subramanian et al., 2009) or lazily (Appel, 1994; Boyapati et al., 2003; Neamtiu et al., 2006; Gurtner, 2011). When updates are performed lazily, the cost of performing the update may be amortised over program execution. The disadvantage of laziness is that it requires monitoring the format of data, which adds steady-state overhead. Eager updates can be implemented without steady state overhead. The disadvantage eagerness is that in order to update all the existing data a stop-the-world must be employed.

Programs that benefit most from dynamic updating are typically structured as long-running event processing loops (Subramanian, 2010). In order to make use of the function body replacement paradigm, long running loops can be extracted into recursive functions, so that a main loop can run to completion using the old code, and that the next time the loop starts (the next call of the loop function) the new code will be used. This can be done automatically (Neamtiu et al., 2006).

Multi-threaded programs provide a particular challenge for existing DSU systems. If updates are performed when the program reaches update point (either explicit or annotated by some monitor) as each thread can reduce independently there is no guarantee that all the threads will reach safe points at the same time. It is possible to block one thread when it reaches a safe point in the hope that the other threads will reach safe points. Other threads could, however, be reliant on the blocked thread to make progress. In some approaches the blocked thread is released after a certain amount of time (Neamtiu and Hicks, 2009). This technique can introduce overhead and delay of an update in the best case, and deadlock to an otherwise live program in the worst case. It is possible to require that no locks are held at update points, in order to try to avoid deadlock (Hayden et al., 2011b). In initial experiments this has not introduced deadlock, but this property has not been proved formally. In addition, stack transformation, if properly designed, can be immediately applied to multi-threaded programs without risk of introducing update-related deadlock (Makris and Bazzi, 2009).

2.3.2 Safety

While it is possible to naively modify any running program in arbitrary ways, the challenge is to do so in a way that leaves a valid updated program, for some definition of valid. The primary definition considered in existing research is type safety (Stoyle, 2006; Bierman et al., 2008; Hjálmtÿsson and Gray, 1998; Makris and Bazzi, 2009; Gurtner, 2011; Subramanian, 2010; Neamtiu and Hicks, 2009; Boyapati et al., 2003; Appel, 1994; Neamtiu et al., 2006; Stoyle et al., 2005). Other safety issues include transaction safety (Neamtiu et al., 2008; Neamtiu and Hicks, 2009), update validity (Gupta et al., 1996), assurance safety by testing (Hayden et al., 2009), and verification of specifications (Hayden et al., 2011a).

2.3.2.1 Type Safety

Type safety is concerned with the prevention of applications from performing illegal operations (those that are disallowed in normal execution). Simple examples of these include performing integer arithmetic on boolean values, or attempting to call a function that doesn't exist. In order to prevent type errors we must co-ordinate how new and old code interact with each other, and with the data. This problem can be simplified by making use of representation consistency, the requirement that data or objects only ever be in one format.

One method of simply showing type safety is to require that any function that is to be updated is not on the call stack when the update is applied. This approach, known as *activeness safety*, neatly partitions old from new code, ensuring that at any one time code and data from only one version are being executed. Activeness safety is one of the most common approaches in DSU (Baumann et al., 2007; Soules et al., 2003; Baumann et al., 2005; Subramanian, 2010; Altekar et al., 2005; Kaashoek and Arnold, 2009) and can be implemented with a simple dynamic check. The requirement that no function that is to be updated is on the stack can lead to an update being delayed for an unbounded time (Subramanian, 2010).

If we permit active functions to be updated, and do not perform stack transformation on the code of these active functions, then the old code could access data that has been transformed into a new format. One solution (known as *con-freeness safety*) is to ensure that the remaining old code (that to be executed before a new function body is called) does not make use of the *format* of the data (referred to as using it *concretely*). Consider the following example:

$$\texttt{let}\, x = (4,5)\, \texttt{in}\, \pi_1(x,x) \tag{2.9}$$

If x were updated to an integer value (here 9) rather than a pair the above code would still be safe, as it does not directly use the fact that x is a pair:

$$\texttt{let}\, x = 9\, \texttt{in}\, \pi_1(x, x) \rightarrow \pi_1(9, 9) \rightarrow 9 \tag{2.10}$$

If the code made direct use of the structure, however, changing to an integer value would lead to an error:

$$\texttt{let}\, x = 9\, \texttt{in}\, \pi_1(\pi_1(x), x) \rightarrow \pi_1(\pi_1(9), 9) \rightarrow \pi_1(9) \rightarrow \texttt{error} \tag{2.11}$$

The original work states that "if code simply passes data around without relying on its representation, then updating that data poses no problem" (Stoyle et al., 2005). We can see that, in Equations 2.9 and 2.10, that the variable x is never used in a way that makes use of its tuple structure (it is only used to construct a new data structure), and hence the usage there is not concrete. In Equation 2.11, however, the tuple structure is used (by the splitting action $\phi_1(x)$), and hence the usage there is concrete. When some code does not use a data type T concretely it is referred to as T-concreteness free, or T-con free. It is possible to automatically detect regions of code that are T-con free for all types that are modified by an update (Stoyle et al., 2005). Any update that occurs in one of these regions will be safe. It is possible to leverage these regions in a multi-threaded setting, waiting until all threads happen to be in a safe region, then pausing and updating (Neamtiu and Hicks, 2009). Preliminary experimental results show that deadlock or update delay does occur, but only rarely.

Some approaches do not require representation consistency, and permit code to interact with old and new versions of data (Hjálmtÿsson and Gray, 1998; Duggan, 2005; Ajmani et al., 2006; Gurtner, 2011). One approach to ensure type-safe data usage is bi-directional transformation functions (Duggan, 2005). These functions transform the data from old to new formats and vice versa. A monitor can then be used to insert transformations when old code uses new data and vice-versa. A similar technique for object-oriented programs is proxy interfaces (Hjálmtÿsson and Gray, 1998; Ajmani et al., 2006; Gurtner, 2011). When an update is introduced, proxies are created for each object, so that any calls to an old class interface are translated into calls to the new interface. The additional indirection required is the primary reason why most designs avoid the use of these techniques.

2.3.2.2 Transaction Safety

In addition to type safety there are other properties that we may wish to preserve, particularly over the update itself. Consider a program that adds a log entry once per iteration of the main loop:

```
1  main
2          f()
3          ...
4          g()
5          h()
6          main()
```

(2.12)

where f() and g() are function calls. In the old code the log entry is performed by function g. Consider an update that changes the bodies of f and g but not the main function itself, and where in the updated program the log entry is performed by f′ and not in g′. Consider also a reduction where the update is applied on line 4. The remaining code in this iteration is:

```
4     g′()
5     h()
6     main′()
```

(2.13)

So in this iteration of the main loop the log entry will not be performed by g, and f has already executed under the old definition. Hence the log entry will be skipped in this iteration. Transaction aids programmer reasoning about update by ensuring that annotated transactional regions appear to behave entirely with old code or entirely with new code (Neamtiu et al., 2008; Neamtiu and Hicks, 2009). They cannot, as is demonstrated here, behave with an old version of one function, and a new version of a different function. Note that when a given update does not update some function, such as h, then this function can be disregarded for the purposes of considering when the update should occur, as it is the same before and after such an update.

2.3.2.3 Update Validity

Gupta et al. introduce the notion of update validity (Gupta et al., 1996). Consider a program point with code P and its state σ. An update consists of some new code P', and a state transformation function s. After an update the program point will be: $[\mathtt{s}(\sigma)]\ P'$. An update is said to be valid if the update program can reduce to some other point $[\sigma'']\ P''$ that is also reachable by reducing the new code with initial state, in effect that an update $[\mathtt{s}(\sigma)]\ P'$ is valid if $[\mathtt{s}(\sigma)]\ P' \to [\sigma'']\ P''$ and if $[\sigma_{\mathtt{init}}]\ P' \to [\sigma'']\ P''$. Gupta et al. show that proving this is, in general, undecidable. They present restrictions to which programs and updates are admissible, that enables proof of update validity. The restrictions are rather severe, and Gupta et al. do not consider type safety issues.

2.3.2.4 Assurance Safety By Testing and Verification

Many developers, rather than building formal models and ensuring that their program adheres to that model, use regression testing as the standard against which the program is measured. This consists of a suite of tests that are run any time a program is changed. The semantics of update itself can introduce unexpected bugs. Performing regression testing on all possible update points, particularly when programs contain concurrency, is prohibitively computationally expensive. In (Hayden et al., 2009), Hayden et. al. show how to reduce this state space into equivalence classes, so that only a representative member need be tested.

This approach is extendible to verification (Hayden et al., 2011a). Hayden et al. define a transformation that merges the old and new versions of a program to create a single program for verification. They describe properties that should be preserved across updates, such as the presence of key-value pairs in a table, even if the internal table format is changed. They prove their merging transformation correct, and show how the DSU specific properties can be checked using off-the-shelf tools that have no understanding of DSU. The is approach incurs an average slow down factor of only four, but can only handle single-threaded programs.

2.3.2.5 No Safety

Some systems do not make any provisions for guaranteeing safety. Edit and continue development for integrated development environments provide some basic functionality to permit the programmer to change code and state during debugging, but makes no guarantees that the changes will be safe (Dmitriev, 2001; Eaddy and Feiner, 2005; Thomas et al., 2010). JRebel is another debugging tool, for Java web applications under development; it intercepts all method invocations and calls the most recent version, but does no state transformation, and hence is not type safe (ZeroTurnaround, 2011). In (Stewart and Chakravarty, 2005), which implements DSU for Haskell, any data migration is done by serialisation, transformation, and re-injection; these are outside the type system and hence not type safe.

One approach where the lack of safety guarantees is intentional is in Erlang (Wikstrom and Williams, 1993). The programming paradigm in Erlang is to assume that various parts of a program (threads, databases, etc.) will fail at some point, and to program defensively. The methodology for update in Erlang is to change the function body and perform any state transformations immediately, and if the remaining old code happens to touch the new data to simply handle the runtime exception that will then be thrown (Erlang, 2010).

2.3.3 Other Considerations

As the aim of DSU is to accommodate unforeseen changes, at unspecified points in execution, it can be difficult to conceptualise how and when updates will occur, and what they will do. Hence ease of use, for programmers, is a key issue in any production DSU approach. Tool support for automatically generating the updates themselves, using the difference between two versions of a program, is included in most DSU approaches aimed at developers (Neamtiu et al., 2006; Gregersen and Jorgensen, 2009; Subramanian, 2010; Thomas et al., 2010; Hayden et al., 2011d). In many systems, however, some programmer intervention is required, particularly in writing non-trivial state transformers (Neamtiu et al., 2006; Hayden et al., 2011d). Some authors argue that reasoning about the behaviour of an update program, particularly when modules or functions are replaced rather than the entire program, is the most difficult cognitive burden for users of DSU (Neamtiu et al., 2008; Hayden et al., 2011d). Others argue that transparency, the ability to use DSU without understanding the semantics or changing coding practices, for example by learning to write state transformation functions, is key to enabling access to DSU in production systems (Gregersen and Jorgensen, 2009). Further research into the practices of software engineers are clearly required.

There are several case studies relevant to DSU in general. The type of changes that occur in practice is the focus of the majority of studies. The incidence of updates that change function or method signatures is surprisingly high, and such changes are included in 10-60% of updates (Neamtiu et al., 2005; Baumann et al., 2007). In addition, the size of updates does not appear to strongly affect the proportion of updates that change signatures (Tempero et al., 2008). The majority of complexity of analyses for DSU, and inefficiencies in implementations, comes from supporting such changes. We consider that, given the proportion of updates that perform changes to signatures, some overhead and cognitive complexity is justified in order to provide update support.

The other case studies focus on dynamic behaviours that can occur when performing DSU. The presence of errors caused by update, and the capability of activeness and con-freeness properties to detect and prevent such errors, is explored in (Hayden et al., 2011c). The majority of errors that were not detected by these properties occurred in updates performed during the initialisation phase of a program. As the majority of updates are assumed to occur during long running main loops, this is not a serious problem. In addition, these properties permitted most updates to safely occur during the main loops, showing that the algorithm was not conservatively preventing updates in the key sections. In (Gregersen and Jorgensen, 2011), interesting, but non-erroneous (at least in terms of type or signature errors) behaviour that occurs when performing DSU in an object oriented system is documented. Transient inconsistency occurs when an updated application is temporally in an unreachable runtime state. An oblivious update is the absence of an expected runtime effect that would have occurred if the system was started from scratch. Phantom objects appear when classes are removed by a dynamic update, but live objects that belong to that class remain. Gregersen et al. suggest various design patterns that can be used to make programs more amenable to updates and to avoid the described phenomena. They particularly emphasise loosely coupled designs, application logic being based on a dynamic interpretation of a program and its current state and not on static assumptions, and the use of declarative registrations of listener objects.

Systems that support DSU introduce overhead above static programs. The majority of the overhead concerns permitting changes to function or object signatures. In non-managed languages, such as those based on C and C++, the indirection necessary to update functions or modules introduces an overhead during steady state (non-updating) execution. Depending on the implementation details this overhead is often between 2-10% in most cases (Neamtiu and Hicks, 2009; Boyapati et al., 2003). In situations where the overhead was higher (10-50%) the overhead appeared only in certain circumstances, and was reduced when a different compiler was used or the program was used in a distributed rather than single-computer setting (Makris and Bazzi, 2009; Neamtiu et al., 2006). The interplay between network semantics, compiler semantics, and DSU would be an interesting path for future research. In managed languages such as Java, much of the indirection for function and method definitions is included by default in the program. It is possible to leverage these facilities to provide DSU with no steady-state overhead verses the static managed language (Subramanian, 2010). When whole program transformation is employed, rather than replacing functions or classes, the function indirection is not required, and DSU can be provide for non-managed languages without any steady-state overhead (Hayden et al., 2011d,b)

Chapter 3

Safety and Liveness Of Concurrent Programs

In this chapter we consider how to represent safety and liveness properties for systems with a variety of side effecting semantics. We explore the communication safety and deadlock freedom properties for asynchronous message passing programs with blocking receives, and provide examples of erroneous programs and their reductions (Section 3.1). We provide a multi-threaded lambda calculus with operators used to access shared state, where the shared state can have an arbitrary semantics (Section 3.2). We show how to use a type and effect system to abstract the impure behaviour of a program, in the vein of Session Types (Section 3.3). We provide a general, inductive description of the communication safety and Deadlock Freedom properties, under the arbitrary shared state semantics (Section 3.4). We show how to re-formulate existing work on Global Session Types, and prove safety and liveness for blocking message passing programs, in an intuitive inductive manner (Section 3.5.1). We also show how to formulate Global Session Types, and prove safety and liveness, for non-blocking message passing, using a similar methodology to that used for blocking message passing (Section 3.5.2). We present our conclusions and describe on our novel contributions in Section 3.6.

$$\begin{aligned}&\texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \texttt{snd}(c_1, 2).\texttt{rcv}(c_3)(x_2 : \textsc{Int}).\\ &\quad \texttt{snd}(c_2, x_2).\texttt{0} \parallel \texttt{rcv}(c_2)(x_3 : \textsc{Bool}).\texttt{rcv}(c_2)(x_3 : \textsc{Int}).\texttt{0}\end{aligned}$$

Figure 3.1: Message Passing Example

3.1 Motivation: Message Passing Programs

In this section we explore communication safety and liveness for asynchronous message passing programs with blocking receives, and provide examples of erroneous programs and their reductions.

3.1.1 Communication Errors

We introduce the concept of Communication Errors in Section 2.2. Intuitively, these occur when a thread receives a value which is not of the type that it expects. The simplest example of this is when the type of a value being sent does not match up with the expected receive type:

$$\begin{array}{ll}
\mathtt{1} & [c \mapsto \emptyset]\ \mathtt{snd}(c, \mathtt{true}).\mathtt{0} \parallel \mathtt{rcv}(c)(x : \textsc{Int}).x + 4 \\
\mathtt{2} & \quad \to [c \mapsto \mathtt{true}]\ \mathtt{0} \parallel \mathtt{rcv}(c)(x : \textsc{Int}).x + 4 \\
\mathtt{3} & \quad \to [c \mapsto \emptyset]\ \mathtt{0} \parallel \mathtt{true} + 4
\end{array} \tag{3.1}$$

The program on line `3` is not typable. The effect of the program on line `1` can be expressed as $c!\langle\textsc{Bool}\rangle \parallel c?(\textsc{Int})$. When written in this form the inconsistency is clear.

The complementarity of the effect of a program is not, however, a sufficient condition to show an absence of Communication Errors; it only is so when there are no messages in the message queues. Existing messages can cause errors even for complementary programs:

$$\begin{array}{ll}
\mathtt{1} & [c \mapsto \mathtt{true}]\ \mathtt{snd}(c, 3).\mathtt{0} \parallel \mathtt{rcv}(c)(x : \textsc{Int}).x + 4 \\
\mathtt{2} & \quad \to [c \mapsto \mathtt{true}, 3]\ \mathtt{0} \parallel \mathtt{rcv}(c)(x : \textsc{Int}).x + 4 \\
\mathtt{3} & \quad \to [c \mapsto 3]\ \mathtt{0} \parallel \mathtt{true} + 4
\end{array} \tag{3.2}$$

The program on line `3` is also not typable. The effect of the program on line `1` is $c!\langle\textsc{Bool}\rangle \parallel c?(\textsc{Bool})$; this effect is complementary, according to the standard definition (Honda, 1993). However, since there are messages in the queue, in order to rule out communication errors and deadlock we cannot consider solely the effect of the code. We must also consider the state of the message queues. We do this by ensuring that the messages in the queue are complementary to the effect of the code, and that after that point the effects are complementary (Bettini et al., 2008). We can show a similar program is safe:

$$\begin{array}{ll}
\mathtt{1} & [c \mapsto 3]\ \mathtt{0} \parallel \mathtt{rcv}(c)(x : \textsc{Int}).x + 4 \\
\mathtt{2} & [c \mapsto \emptyset]\ \mathtt{0} \parallel 3 + 4 \\
\mathtt{3} & [c \mapsto \emptyset]\ \mathtt{0} \parallel 7
\end{array} \tag{3.3}$$

1 $[\sigma_\emptyset]$ $\mathtt{rcv}(c_1)(x_1 : \textsc{Int}).\mathtt{snd}(c_2, \mathtt{true}).\mathtt{snd}(c_3, x_1 + 1).\mathtt{0} \parallel$
$\mathtt{snd}(c_1, 2).\mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel$
$\mathtt{rcv}(c_2)(x_3 : \textsc{Bool}).\mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

2 $[c_1 \mapsto 2]$ $\mathtt{rcv}(c_1)(x_1 : \textsc{Int}).\mathtt{snd}(c_2, \mathtt{true}).\mathtt{snd}(c_3, x_1 + 1).\mathtt{0} \parallel$
$\mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel \mathtt{rcv}(c_2)(x_3 : \textsc{Bool}).\mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

3 $[\sigma_\emptyset]$ $\mathtt{snd}(c_2, \mathtt{true}).\mathtt{snd}(c_3, 2 + 1).\mathtt{0} \parallel \mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel$
$\mathtt{rcv}(c_2)(x_3 : \textsc{Bool}).\mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

4 $[c_2 \mapsto \mathtt{true}]$ $\mathtt{snd}(c_3, 2 + 1).\mathtt{0} \parallel \mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel$
$\mathtt{rcv}(c_2)(x_3 : \textsc{Bool}).\mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

5 $[\sigma_\emptyset]$ $\mathtt{snd}(c_3, 2 + 1).\mathtt{0} \parallel \mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel \mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

6 $[c_3 \mapsto 3]$ $\mathtt{0} \parallel \mathtt{rcv}(c_3)(x_2 : \textsc{Int}).\mathtt{snd}(c_2, x_2).\mathtt{0} \parallel \mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

7 $[\sigma_\emptyset]$ $\mathtt{0} \parallel \mathtt{snd}(c_2, 3).\mathtt{0} \parallel \mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

8 $[c_2 \mapsto 3]$ $\mathtt{0} \parallel \mathtt{0} \parallel \mathtt{rcv}(c_2)(x_3 : \textsc{Int}).\mathtt{0}$

9 $[\sigma_\emptyset]$ $\mathtt{0} \parallel \mathtt{0} \parallel \mathtt{0}$

Figure 3.2: Safe Blocking Semantics Reduction

The effect of this program is $\mathtt{0} \parallel c?(\textsc{Int})$. Here, the $c?(\textsc{Int})$ is complementary to the Int value in the message queue.

The influence of the semantics of the shared state on Communication Safety can be illustrated using the program in Figure 3.1. Under a semantics with blocking receives there is no reduction sequence that contains Communication Errors; one such safe reduction is shown in Figure 3.2.

One alternate semantics for message passing has non-blocking receives; when a receive action is performed on an empty queue the action does not block. Normally, when performing a receive action on an empty queue, some specific error value would be returned, and the control flow would then handle the fact that a value of the expected type has not been returned. For the sake of the next example, however, we assume such a receive returns a default value of the expected type. Under such a semantics we can obtain a Communication Error, as in Figure 3.3. As the second thread does not synchronise by receiving the value from the first thread, it can bypass its receive and perform its send to the third thread ($\mathtt{snd}(c_2, x_2).\ldots$) before the first thread performs its send to the third thread ($\mathtt{snd}(c_2, \mathtt{true}).\ldots$). This leads to the third thread receiving an Int when it is expecting a Bool. This exemplifies how synchronisation behaviour is key to proving safety in many circumstances.

$$\begin{array}{ll}
1 & [\sigma_\emptyset]\ \texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \\
 & \quad \texttt{snd}(c_1, 2).\texttt{rcv}(c_3)(x_2 : \textsc{Int}).\texttt{snd}(c_2, x_2).\texttt{0} \parallel \\
 & \quad \texttt{rcv}(c_2)(x_3 : \textsc{Bool}).\texttt{rcv}(c_2)(x_3 : \textsc{Int}).\texttt{0} \\
2 & [c_1 \mapsto 2]\ \texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \\
 & \quad \texttt{rcv}(c_3)(x_2 : \textsc{Int}).\texttt{snd}(c_2, x_2).\texttt{0} \parallel \texttt{rcv}(c_2)(x_3 : \textsc{Bool}).\texttt{rcv}(c_2)(x_3 : \textsc{Int}).\texttt{0} \\
3 & [c_1 \mapsto 2]\ \texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \texttt{snd}(c_2, 0).\texttt{0} \parallel \\
 & \quad \texttt{rcv}(c_2)(x_3 : \textsc{Bool}).\texttt{rcv}(c_2)(x_3 : \textsc{Int}).\texttt{0} \\
4 & [c_1 \mapsto 2, c_2 \mapsto 0]\ \texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \texttt{0} \parallel \\
 & \quad \texttt{rcv}(c_2)(x_3 : \textsc{Bool}).\texttt{rcv}(c_2)(x_3 : \textsc{Int}).\texttt{0} \\
5 & [c_1 \mapsto 2]\ \texttt{rcv}(c_1)(x_1 : \textsc{Int}).\texttt{snd}(c_2, \texttt{true}).\texttt{snd}(c_3, x_1 + 1).\texttt{0} \parallel \texttt{0} \parallel \texttt{error}
\end{array}$$

Figure 3.3: Non Blocking Semantics Gives Rise to Communication Error

3.1.2 Deadlock

Deadlock, due to message passing, occurs when a communication action is blocked indefinitely. If no communication actions can block then we can trivially show an absence of deadlock. As we show in Example 3.3, however, intuition about which programs are safe can be incorrect when programs perform less synchronisation than we expect; in the above examples, synchronisation occurs when individual threads block temporarily at receives until messages become available. We therefore consider it important to work in systems which have some form of synchronisation, and which hence can deadlock.

The simplest example of deadlock is a program where one process expects a value to be sent and the others do not:

$$\begin{array}{ll}
1 & [c \mapsto \emptyset]\ \texttt{0} \parallel \texttt{rcv}(c)(x : \textsc{Int}).x + 4 \\
2 & \quad \not\to
\end{array} \tag{3.4}$$

Here the two processes do not have complementary behaviour, and will obviously deadlock. Similarly to Example 3.2, complementary behaviour is not a sufficient condition to rule out deadlock when there are existing messages in the queue:

$$\begin{array}{ll}
1 & [c_1 \mapsto l_2]\ \texttt{case}\ c_1\ \texttt{of}\ \{l_1 \mapsto \texttt{snd}(c_2, 3).\texttt{0}, l_2 \mapsto \texttt{0}\} \parallel \texttt{sel}(c_1, l_1).\texttt{rcv}(c_2)(x : \textsc{Int}).\texttt{0} \\
2 & [c_1 \mapsto l_2, l_1]\ \texttt{case}\ c_1\ \texttt{of}\ \{l_1 \mapsto \texttt{snd}(c_2, 3).\texttt{0}, l_2 \mapsto \texttt{0}\} \parallel \texttt{rcv}(c_2)(x : \textsc{Int}).\texttt{0} \\
3 & [c_1 \mapsto l_1]\ \texttt{0} \parallel \texttt{rcv}(c_2)(x : \textsc{Int}).\texttt{0} \\
4 & \quad \not\to
\end{array} \tag{3.5}$$

In this case existing messages influence the control flow of the program, leading to the second process expecting the l_1 service whilst the first process provides the l_2 service.

$$
\begin{array}{lcll}
t & ::= & & \textit{Terms} \\
 & \mid & v & \text{Values} \\
 & \mid & \underline{X} & \text{Recursion Variables} \\
 & \mid & t\,t & \text{Application} \\
 & \mid & (\alpha(\widetilde{v}), T) & \text{State Access} \\
 & \mid & \texttt{if}\, t\, \texttt{then}\, t\, \texttt{else}\, t & \text{Conditional} \\
 & \mid & \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T \mapsto t}\} & \text{Case Split} \\
 & \mid & \texttt{error} & \text{Error} \\
 & & & \\
v & ::= & & \textit{Values} \\
 & \mid & n & \text{Integers} \\
 & \mid & b & \text{Booleans} \\
 & \mid & () & \text{Unit} \\
 & \mid & r & \text{Resources} \\
 & \mid & x & \text{Variables} \\
 & \mid & l & \text{Service Labels} \\
 & \mid & \texttt{rec}\, \underline{X}(x : T).t & \text{Recursive Functions} \\
 & & & \\
P & ::= & & \textit{Process Threads} \\
 & \mid & t & \text{Single Term Thread} \\
 & \mid & P \parallel P & \text{Parallel Compostition}
\end{array}
$$

Figure 3.4: Language

We conclude that the safety and liveness of a program depends on: 1) the communication actions of the program, 2) the messages in the message queues, and 3) the semantics of the communication actions.

3.2 Language Definitions

We define our language in Figure 3.4 and our operational semantics in Figure 3.5. Values consist of integers, booleans, the unit value, variables, resources, and recursive functions. Resources are handles into the runtime state, such as a channel name c in previous examples. Labels are used to provide unique service labels. Recursive functions $\texttt{rec}\,\underline{X}(x : T).t$ consist of a recursion variable $\underline{X}$, a formal parameter x, the expected type T of the variable (we define types in Section 3.3), and the body of the function t. Function application is denoted as $t_1\,t_2$. The term $(\texttt{rec}\,\underline{X}(x : T).t)\,v$ reduces to $t[v/x][\texttt{rec}\,\underline{X}(x : T).t_2/\underline{X}]$, assuming $v\colon T$, and $\texttt{error}$ otherwise. We make use of an eager reduction strategy, and hence use $t_1;\, t_2$ as syntactic sugar for $(\texttt{rec}\,\underline{X}(x : T).t_2)\,t_1$, where x and $\underline{X}$ are not free in t_2. We also make use of the syntactic sugar $\lambda x : T.t$ for $\texttt{rec}\,\underline{X}(x : T).t$ when $\underline{X}$ is not free in t.

Accesses to the shared state are denoted using $(\alpha(\widetilde{v}), T)$. This denotes that action α is performed using parameters $\widetilde{v}$, and this access should return a value of type T.

$$\frac{[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'}{[\sigma]\ t_1\,t_2 \xrightarrow{\gamma} [\sigma']\ t_1'\,t_2} \qquad \frac{[\sigma]\ t_2 \xrightarrow{\gamma} [\sigma']\ t_2'}{[\sigma]\ v\,t_2' \xrightarrow{\gamma} [\sigma']\ v\,t_2'}$$

$$\frac{}{[\sigma]\ \texttt{rec}\,\underline{X}(x:T).t\,v \xrightarrow{\tau} [\sigma]\ t[\texttt{rec}\,\underline{X}(x:T).t/\underline{X}][v/x]}$$

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v}))\colon T}{[\sigma]\ (\alpha(\widetilde{v}),T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \sigma(\alpha(\widetilde{v}))} \qquad \frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v}))\colon T' \neq T}{[\sigma]\ (\alpha(\widetilde{v}),T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}}$$

$$\frac{}{[\sigma]\ \texttt{if}\,\texttt{true}\,\texttt{then}\,t_2\,\texttt{else}\,t_3 \xrightarrow{\tau} [\sigma]\ t_2} \qquad \frac{}{[\sigma]\ \texttt{if}\,\texttt{false}\,\texttt{then}\,t_2\,\texttt{else}\,t_3 \xrightarrow{\tau} [\sigma]\ t_3}$$

$$\frac{[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'}{[\sigma]\ \texttt{if}\,t_1\,\texttt{then}\,t_2\,\texttt{else}\,t_3 \xrightarrow{\gamma} [\sigma']\ \texttt{if}\,t_1'\,\texttt{then}\,t_2\,\texttt{else}\,t_3}$$

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T_i \qquad T_i \in \widetilde{T}}{[\sigma]\ \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ t_i} \qquad \frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T' \qquad T' \notin \widetilde{T}}{[\sigma]\ \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}}$$

$$\frac{[\sigma]\ P_1 \xrightarrow{\gamma} [\sigma']\ P_1'}{[\sigma]\ P_1 \,||\, P_2 \xrightarrow{\gamma} [\sigma']\ P_1' \,||\, P_2} \qquad \frac{[\sigma]\ P_2 \xrightarrow{\gamma} [\sigma']\ P_2'}{[\sigma]\ P_1 \,||\, P_2 \xrightarrow{\gamma} [\sigma']\ P_1 \,||\, P_2'}$$

Figure 3.5: Operational Semantics

State accesses are not permitted to return values containing resources. Were we to permit resources to be passed around, then channels could be passed around, and we would need to consider session typing techniques for delegation. In a type and effect formulation, this would require dependent typing within the effect system, which whilst possible would significantly complicate our current analysis. Examples of state accesses include $(\texttt{snd}(c,3), \textsc{Unit})$ and $(\texttt{rcv}(c), \textsc{Int})$ which denote the communication actions of $\texttt{snd}(c,3)$ and $\texttt{rcv}(c)(x : \textsc{Int})$ respectively, but not their continuations or the variable binding for `rcv`.

The conditional operator is standard. Service branching is provided using the case split operator $\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}$. This operator performs the communication action $\alpha(\widetilde{v})$, as defined above, and then does a case split on the type of the result. If the value returned is not of an expected type then the case split reduces to `error`. We denote errors using the `error` term; errors cannot reduce, in and of themselves.

We parameterise our work on a *resource model*, which is a collection of *states* ranged over by σ, and a function that maps a state and a label to a return value and another state: we write $\sigma(\alpha(\widetilde{v})) = v$ and $\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$ to denote this map. The resource model is a generalised model, and hence can be instantiated with either finite or infinite models. In particular, it is not restricted to queues, though in this thesis we will only consider examples that use queues.

$[\![\texttt{snd}(c,e).t]\!]$	$\overset{\text{def}}{=}$	$(\lambda x : T.(\texttt{snd}(c,x), \textsc{Unit})\ [\![e]\!]);\ [\![t]\!]$	Sends
$[\![\texttt{rcv}(c)(x : T).t]\!]$	$\overset{\text{def}}{=}$	$(\lambda x : T.[\![t]\!])\ (\texttt{rcv}(c), T)$	Receives
$[\![\texttt{if}\, e\, \texttt{then}\, t\, \texttt{else}\, t]\!]$	$\overset{\text{def}}{=}$	$\texttt{if}\ [\![e]\!]\ \texttt{then}\ [\![t]\!]\ \texttt{else}\ [\![t]\!]$	Conditionals
$[\![\texttt{sel}(c,l).t]\!]$	$\overset{\text{def}}{=}$	$(\texttt{snd}(c,l), \textsc{Unit});\ [\![t]\!]$	Service Selection
$[\![\texttt{case}\ c\ \texttt{of}\ \{\widetilde{l \mapsto t}\}]\!]$	$\overset{\text{def}}{=}$	$\texttt{case}\,\texttt{rcv}(c)\,\texttt{in}\,\{\widetilde{\texttt{Lab}\, l \mapsto [\![t]\!]}\}$	Service Branching
$[\![\mu\underline{X}.t]\!]$	$\overset{\text{def}}{=}$	$\texttt{rec}\ \underline{X}(x : \textsc{Unit}).[\![t]\!]$	Recursion, where x is not free in t
$[\![\underline{X}]\!]$	$\overset{\text{def}}{=}$	$\underline{X}\,()$	Recursion Variable
$[\![\texttt{error}]\!]$	$\overset{\text{def}}{=}$	$\texttt{error}$	Errors
$[\![e]\!]$	$\overset{\text{def}}{=}$	$\ldots$	Typical λ-calculus embedding of numbers, etc.
$[\![P \parallel P]\!]$	$\overset{\text{def}}{=}$	$[\![P]\!] \parallel [\![P]\!]$	Parallel Threads

$$\frac{}{\sigma[c \mapsto q] \xrightarrow{\texttt{snd}(c,v)} \sigma[c \mapsto (q,v)]} \qquad \frac{}{\sigma[c \mapsto (v,q)] \xrightarrow{\texttt{rcv}(c)} \sigma[c \mapsto q]}$$

$$\sigma(\texttt{snd}(c,v)) \overset{\text{def}}{=} () \qquad \sigma[c \mapsto v,q](\texttt{rcv}(c)) \overset{\text{def}}{=} v$$

Figure 3.6: Blocking Message Passing Calculus Embedding

We can embed the message passing calculus presented in Section 2.2 in our language. We describe this embedding in Figure 3.6; it is relatively straightforward, using functions to represent binding of received values. Recursion variables are mapped to an application of unit to the recursion variable.

3.3 Behavioural Abstraction

We use effects as an abstraction of the actions on the shared state. We describe this behaviour in both thread-local and global (parallel) settings. We also present an effect system that determines the effect of a program. In order to provide our safety and liveness guarantees, however, we need to do more than determine the effect of a program. We need to place some restrictions on which programs are valid. We describe these restrictions in Sections 3.4.

3.3.1 Parallel and Local Session Types

In Figure 3.7 we present the syntax of our types and effects.

Types consist of base types, function types, and two nominal types (those where the type is defined by the singleton value which has said type). A function type $T_1 \xrightarrow{\varphi} T_2$

φ	$::=$		*Local Effects*
	$\mid$	$(\alpha(\widetilde{T}), T)$	State Access
	$\mid$	$\varphi;\ \varphi$	Sequence
	$\mid$	$\varphi \oplus \varphi$	Internal Choice
	$\mid$	$\varphi \& \varphi$	External Choice
	$\mid$	$\mu \underline{X}.\varphi$	Recursion
	$\mid$	$\underline{X}$	Recursion Variable
	$\mid$	ϵ	Empty Effect
	$\mid$	`error`	Error

T	$::=$		*Types*
	$\mid$	BOOL	Booleans
	$\mid$	INT	Integers
	$\mid$	UNIT	Units
	$\mid$	$\mathsf{Res}\, r$	Resources
	$\mid$	$\mathtt{Lab}\, l$	Labels
	$\mid$	$T \xrightarrow{\varphi} T$	Functions
Φ	$::=$		*Parallel Effects*
	$\mid$	φ	Local effect
	$\mid$	$\Phi \parallel \Phi$	Parallel Composition

Figure 3.7: Types, and Local and Parallel Effects

is annotated with the latent effect φ of the function's body (as in Section 2.1). This denotes that when an argument is applied to the function, the body of the function will reduce with an effect described by φ. To aid our later analyses we wish to distinguish all resources and labels at the type level. We hence use nominal types for resources and labels; a resource r has type $\mathsf{Res}\, r$, and a label l has type $\mathtt{Lab}\, l$, respectively.

Local Effects describe the structure of actions performed by a single thread on the shared state. An abstract action $(\alpha(\widetilde{T}), T)$ denotes performing an action α, where the types of the parameters to the action are $\widetilde{T}$, and the expected type of the return value is T. For example, the action $(\mathtt{snd}(c, 3), \textsc{Unit})$ is abstracted using $(\mathtt{snd}(\mathsf{Res}\, c, \textsc{Int}), \textsc{Unit})$. Sequencing, recursion, empty effects, and errors are standard. Internal choice $\varphi_1 \oplus \varphi_2$ denotes that a program can proceed with either the behaviour described in φ_1 or the behaviour described in φ_2, and the choice will be made within the given process in an opaque manner. We will use internal choice to describe the behaviour of `if` statements. External choice $\varphi_1 \& \varphi_2$ denotes that a program can proceed with either the behaviour described in φ_1 or that described in φ_2, but that in order to choose a specific path the first action of that path must be reducible, and not blocked, at the time the choice is made. Parallel Effects are simply Local Effects in parallel with each other.

We make use of an equivalence relation over effects, which is the least relation defined structurally over Parallel Effects using:

$$\epsilon;\ \varphi \equiv \varphi \equiv \varphi;\ \epsilon \qquad \mu \underline{X}.\varphi \equiv \varphi[\mu \underline{X}.\varphi / \underline{X}] \tag{3.6}$$

3.3.2 Typing Rules

In Figure 3.8 we present our type and effect system. A typing judgement $\varphi \wr \Gamma \vdash t\colon T$ denotes that, under typing assumptions Γ the communication behaviour of some code t is represented by the effect φ, and the code will return a value of type T. A typing judgement for parallel processes is similarly defined, without the typing environment or the return type. We make use of a type system for simple expressions, whose judgement $t\colon T$ denotes that t is a term, that performs no actions on the shared state, whose return type is T.

Values are abstracted to empty effects. Resources and labels are nominally typed. Variables are typed using the typing assumptions. Actions are abstracted using the types of the parameters, and we require that an action does not return a resource type. Recursive functions are typed under the assumption that the variable x has type T_1, and that the function itself has type $T_1 \xrightarrow{X} T_2$. An action $(\alpha(\widetilde{v}), T)$, that reduces the shared state using label $\alpha(\widetilde{v})$ and expects a return value of type T, assumes that the value returned will be of the expected type, and that it will not be a resource. Note that at this stage we have no guarantee that the return value actually will be of the expected type. In order to ensure that communication errors do not occur, such as that in Example 3.2, we need to verify that no such errors exist. We describe this verification in Section 3.4.

Recursive functions are abstracted using a function type, with a latent recursive effect. When typing the body of the function we assume that the recursion variable has the type of a function from T_1 to T_2, with the latent effect $\underline{X}$ to denote the recursion. We require that x is not free in Γ, T_1, T_2, or φ. This prevents the function from being passed a resource variable as a parameter, and effectively prohibits functions from having polymorphic effect dependent on the function parameter. Application is abstracted as a sequence of effects, firstly that of reducing the left hand side to a value, then that of reducing the right hand side to a value, then that of reducing the function body.

Conditional terms are abstracted as the effect of the term being split over, followed by the internal choice between the effects of the consequent and the alternative. We require that the predicate not access the shared state. A case split $\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}_M$ is abstracted as the external choice between, for each $m \in M$, performing action $\alpha(\widetilde{T})$ and receiving a value of type T_m, followed by the effect of the continuation t_m. As such $\&\varphi_i$ is syntactic sugar for $\varphi_1 \& \ldots \& \varphi_n$ where $I = \{1, \ldots, n\}$. We also assume that none of the values received are resources. Threads are typed using an empty type environment. Parallel threads are abstracted as the parallel composition of their constituent effects.

3.3.3 Semantics of Effects

As the type and effect system rules out simple type errors such as $4 + \texttt{true}$, type errors can only be introduced by the reception of a value which is not of the type that a

$$\frac{}{\epsilon \wr \Gamma \vdash n \colon \textsc{Int}} \qquad \frac{}{\epsilon \wr \Gamma \vdash b \colon \textsc{Bool}} \qquad \frac{}{\epsilon \wr \Gamma \vdash () \colon \textsc{Unit}} \qquad \frac{}{\epsilon \wr \Gamma \vdash r \colon \mathsf{Res}\, r}$$

$$\frac{}{\epsilon \wr \Gamma \vdash x \colon \Gamma(x)} \qquad \frac{}{\epsilon \wr \Gamma \vdash l \colon \mathsf{Lab}\, l} \qquad \frac{\epsilon \wr \Gamma \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma \vdash (\alpha(\widetilde{v}), T) \colon T}$$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t \colon T_2 \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \mathtt{rec}\, \underline{X}(x : T_1).t \colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma \vdash t_2 \colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma \vdash t_1\, t_2 \colon T_1}$$

$$\frac{\begin{array}{c}\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Bool} \\ \varphi_2 \wr \Gamma \vdash t_2 \colon T \quad \varphi_3 \wr \Gamma \vdash t_3 \colon T\end{array}}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \Gamma \vdash \mathtt{if}\, t_1\, \mathtt{then}\, t_2\, \mathtt{else}\, t_3 \colon T}$$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \Gamma \vdash \mathtt{case}\, \alpha(\widetilde{v})\, \mathtt{in}\, \{\widetilde{T \mapsto t}\} \colon T}$$

$$\frac{\varphi \wr \Gamma \vdash t \colon T \qquad \varphi \equiv \varphi'}{\varphi \wr \Gamma \vdash t \colon \varphi'} \qquad \frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi} \qquad \frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2}$$

Figure 3.8: Typing Rules

program expects. In other words, the only source of type errors is actions performed on the shared state; we can therefore prove an absence of type errors by considering the actions on the shared state alone. In order to reason about how the effect of a program reduces, in the presence of some shared state, we also make use of an *abstract resource model* which abstracts away from particular values associated with resource states and provides a representation of the resource types. An abstract resource model is a collection of abstract states ranged over by Σ and a function that maps a state and an abstract label to a type and another abstract state: we write $\Sigma(L) = T$ and $\Sigma \xrightarrow{L} \Sigma'$ as above. Abstract actions are of the form $\alpha(\widetilde{T})$, so that a label $\alpha(\widetilde{v})$ corresponds to a unique abstract label $\alpha(\widetilde{T})$ via simple typing of values. We assume the fact that we can relate the resource model and abstract resource model via an abstraction map Λ with the property that $\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$ implies $\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma')$ and $\sigma(\alpha(\widetilde{v})) \colon \Lambda(\sigma)(\alpha(\widetilde{T}))$. We include a semantics for effects and abstract models in Figure 3.9, with a definition of the label abstraction rule in Figure 3.10.

If an action returns a value of the expected type ,then that action reduces to the empty effect. If that action does not return a value of the expected type that action reduces to an error. Sequential composition and internal choice are standard. External choice can only reduce to one of the branches when that branch can also reduce (i.e. it is not blocked). When both options are enabled, if one can reduce without causing an error, that reduction will be chosen. If both can reduce without error then the path

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\ (\alpha(\widetilde{T}), T) \rightarrow [\Sigma']\ \epsilon} \qquad \frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) \neq T}{[\Sigma]\ (\alpha(\widetilde{T}), T) \rightarrow [\Sigma']\ \texttt{error}}$$

$$\frac{[\Sigma]\ \varphi_1 \xrightarrow{\gamma} [\Sigma']\ \varphi_1'}{[\Sigma]\ \varphi_1 ; \varphi_2 \xrightarrow{\gamma} [\Sigma']\ \varphi_1' ; \varphi_2} \qquad \frac{i \in 1, 2}{[\Sigma]\ \varphi_1 \oplus \varphi_2 \rightarrow [\Sigma]\ \varphi_i}$$

$$\frac{\exists \varphi_i', \alpha_i.\ [\Sigma]\ \varphi_i \xrightarrow{\alpha_i(\widetilde{T})} [\Sigma_i']\ \varphi_i' \wedge \texttt{error} \in \varphi_i'}{[\Sigma]\ \varphi_1 \& \varphi_2 \xrightarrow{\alpha_i(\widetilde{T})} [\Sigma_i']\ \varphi_i'}$$

$$\frac{\forall \varphi_i', \alpha_i.\ [\Sigma]\ \varphi_i \xrightarrow{\alpha_i(\widetilde{T})} [\Sigma_i']\ \varphi_i' \Rightarrow \texttt{error} \in \varphi_i'}{[\Sigma]\ \varphi_1 \& \varphi_2 \xrightarrow{\alpha_j(\widetilde{T})} [\Sigma_j']\ \varphi_j'}$$

$$\frac{}{[\Sigma]\ \mu\underline{X}.\varphi \rightarrow [\Sigma]\ \varphi[\mu\underline{X}.\varphi/\underline{X}]} \qquad \frac{\Phi \equiv \Phi_1 \quad [\Sigma]\ \Phi_1 \rightarrow [\Sigma']\ \Phi_2 \quad \Phi_2 \equiv \Phi'}{[\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi'}$$

$$\frac{[\Sigma]\ \Phi_1 \xrightarrow{\gamma} [\Sigma']\ \Phi_1'}{[\Sigma]\ \Phi_1 \,||\, \Phi_2 \xrightarrow{\gamma} [\Sigma']\ \Phi_1' \,||\, \Phi_2} \qquad \frac{[\Sigma]\ \Phi_2 \xrightarrow{\gamma} [\Sigma']\ \Phi_2'}{[\Sigma]\ \Phi_1 \,||\, \Phi_2 \xrightarrow{\gamma} [\Sigma']\ \Phi_1 \,||\, \Phi_2'}$$

Figure 3.9: Effect Semantics

$$\frac{\epsilon \wr \emptyset \vdash v_i : T_i}{\alpha(\widetilde{T}) \vdash \alpha(\widetilde{v})}$$

Figure 3.10: Label Abstraction

is chosen non-deterministically. If both can reduce with an error then the path is also chosen non-deterministically. Recursion unfolding, equivalence reduction, and parallel reduction are standard.

One contribution in this area is the explicit definition of what type should be returned for a given shared state access. Whilst this is currently done in Session Typing analyses for receive actions, we extend it to all actions. One use for being able to return, and handle, different types for the same action permits us to provide information about the shared state and how it has responded to the action. This is particularly used in our handling of the control flow of non-blocking message passing (Section 3.5.2).

3.4 General Formulation

In order to provide safety and liveness guarantees we need to do more than determine the effect of a program. We must place some restrictions on which programs are valid, in

$$
\begin{array}{lll}
\mathcal{G}^0(\Sigma,\Phi) & \overset{\text{def}}{=} & \texttt{true} \\
\mathcal{G}^{k+1}(\Sigma,\Phi) & \overset{\text{def}}{=} & \left[\begin{array}{l} [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow \mathcal{G}^k(\Sigma',\Phi') \\ \quad \wedge [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \vee \Phi = \Pi\,\epsilon \\ \quad \wedge \nexists \texttt{error} \in \Phi \end{array}\right. \\
\quad \text{where } k > 0 & &
\end{array}
$$

$$\mathcal{G}(\Sigma,\Phi) = \bigwedge_{k\in\mathcal{N}} \mathcal{G}^k(\Sigma,\Phi)$$

Figure 3.11: Global Compatibility

order to rule out errors and deadlock, such as those discussed in Section 3.1. We define a program point to consist of an effect Φ and a shared state abstraction Σ (or a program and some shared state, respectively). A program point is safe and deadlock free if it does not contain an error, if all its reductions result in safe and deadlock free program points, and if it can perform at least one reduction (or consists of empty effects). We formalise these using the function in Figure 3.11. We refer to this property as *Global Compatibility*, and denote a globally compatible program point as $\mathcal{G}(\Sigma,\Phi)$. We describe this formalisation in more detail as follows:

$$\nexists \texttt{error} \in \Phi \tag{3.7}$$

This requires that no errors exist in a globally compatible program point.

$$[\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow \mathcal{G}(\Sigma',\Phi') \tag{3.8}$$

This requires that for any reduction which can occur at this point that the reduced program must also be globally compatible.

$$[\Sigma]\ \Phi \to [\Sigma']\ \Phi' \vee \Phi \equiv \Pi\,\epsilon \tag{3.9}$$

This requires that either the effect can reduce, or it is equivalent to empty effects.

The predicate can alternatively be characterised as the greatest fixed point of the following function, defined over the state space S of all possible pairs (Σ,Φ) of abstract states and parallel effects:

$$g(S) = \left\{ \begin{array}{lll} (\Sigma,\Phi) & | & \forall \Sigma',\Phi'.[\Sigma]\Phi \to [\Sigma']\Phi' \Rightarrow (\Sigma',\Phi') \in S \text{ and} \\ & & (\exists \Sigma',\Phi'.\,[\Sigma]\Phi \to [\Sigma']\Phi' \text{ or } \Phi \equiv \Pi\,\epsilon) \text{ and } \nexists \texttt{error} \in \Phi \end{array} \right\} \tag{3.10}$$

This is a monotone function on the powerset lattice of pairs of abstract states and effects. It has a greatest fixed point νg (which is just a set of tuples). Using the Knaster-Tarski fixed point theorem, we know that the predicate $\mathcal{G}(\Sigma,\Phi)$ is defined if and only if $(\Sigma,\Phi) \in \nu g$.

3.4.1 Key Lemmas

Using the definition of Global Compatibility we can proceed to prove some key lemmas.

3.4.1.1 Subject Reduction

If a term and its accompanying shared state's abstraction is globally compatible then we know that:

$$\boxed{[\Sigma]}\ \varphi \rightarrow \boxed{[\Sigma']}\ \varphi' \Rightarrow \texttt{error} \notin \varphi' \tag{3.11}$$

This is as, if $\mathcal{G}(\Sigma, \varphi)$, then we know that $\mathcal{G}(\Sigma', \varphi')$ and that if $\mathcal{G}(\Sigma', \varphi')$ then $\texttt{error} \notin \varphi'$. Given that there are no type rules for `error` we know that all well typed programs are error free. We can use Equation 3.11 to show that a single reduction results in a well typed reduced term, and that the effect of the original program point can reduce to the effect of the new program point.

Lemma 3.1. *Expression Subject Reduction*

For all $\varphi, \varphi'', t, t', T, \sigma, \sigma''$ if $\varphi \wr \emptyset \vdash t \colon T$, and $\boxed{[\sigma]}\ t \xrightarrow{\gamma} \boxed{[\sigma']}\ t'$ and if whenever $\gamma' \vdash \gamma$ and $\boxed{[\Lambda(\sigma)]}\ \varphi \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma'')]}\ \varphi''$ then $\texttt{error} \notin \varphi''$, then there exists a φ' such that typing judgement $\varphi' \wr \emptyset \vdash t' \colon T$ holds.

Proof. Suppose the hypotheses. We then prove the conclusions simultaneously by induction over the derivation of $\boxed{[\sigma]}\ t \xrightarrow{\gamma} \boxed{[\sigma']}\ t'$. The key cases are as follows.

Consider the case where the last reduction rule used in the derivation is:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T}{\boxed{[\sigma]}\ (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} \boxed{[\sigma']}\ \sigma(\alpha(\widetilde{v}))}$$

As $t = (\alpha(\widetilde{v}), T)$ and $\varphi \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T$, then by the type rules we know that the following holds:

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

where $\varphi \equiv (\alpha(\widetilde{T}), T)$. Given that $\gamma = \alpha(\widetilde{v})$, by the definition of $\gamma' \vdash \gamma$ we know that then any relevant γ' must be $\alpha(\widetilde{T})$ where $\widetilde{v} \colon \widetilde{T}$. If $\boxed{[\Lambda(\sigma)]}\ (\alpha(\widetilde{T}), T) \xrightarrow{\gamma'}_F \boxed{[\Lambda(\sigma'')]}\ \varphi''$ then it will be by one of the following two rules:

$$\frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) = T}{\boxed{[\Lambda(\sigma)]}\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} \boxed{[\Lambda(\sigma')]}\ \epsilon} \qquad \frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) \neq T}{\boxed{[\Lambda(\sigma)]}\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} \boxed{[\Lambda(\sigma')]}\ \texttt{error}}$$

As we assume that $\texttt{error} \notin \varphi''$ then we must use the first rule, and hence have that $\varphi'' = \epsilon$ and that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T$. By our assumptions about the definition of the state projection function we know that $\sigma(\alpha(\widetilde{v}))\colon T$. Then by Lemma A.9 we know that $\epsilon \wr \emptyset \vdash \sigma(\alpha(\widetilde{v}))\colon T$ holds, as required.

Consider the case where the last reduction rule used in the derivation is:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T' \neq T}{[\sigma]\ (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}}$$

As $t = (\alpha(\widetilde{v}), T)$ and $\varphi \wr \emptyset \vdash (\alpha(\widetilde{v}), T)\colon T$, then by the type rules we know that the following holds:

$$\frac{\epsilon \wr \emptyset \vdash v_i\colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T)\colon T}$$

where $\varphi \equiv (\alpha(\widetilde{T}), T)$. Given that $\gamma = \alpha(\widetilde{v})$, by the definition of $\gamma' \vdash \gamma$ we know that then any relevant γ' must be $\alpha(\widetilde{T})$ where $\widetilde{v}\colon \widetilde{T}$. If $[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\gamma'}_F [\Lambda(\sigma'')]\ \varphi''$ then it will be by one of the following two rules:

$$\frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) = T}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \epsilon} \qquad \frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) \neq T}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \texttt{error}}$$

Consider when the first rule is used. Then we must have that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T$. However, by the reduction rule being considered we have that $\Lambda(\sigma)(\alpha(\widetilde{T})) \neq T$, as $\sigma(\alpha(\widetilde{v}))\colon T'' \neq T$, and by our assumptions we must have T'' be the same as $\Lambda(\sigma)(\alpha(\widetilde{v}))$. Hence we have a contradiction.

Consider when the second rule is used. Then we have that $[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \texttt{error}$. By our assumptions, however, we do not consider any such reductions. Hence we have a contradiction.

As our hypotheses are not fulfilled in this example, indeed they provide a contradiction, then we can disregard the $[\sigma]\ (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}$ reduction.

Consider the case where the last reduction rule used in the derivation is:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T_i \qquad T_i \in \widetilde{T}}{[\sigma]\ \mathsf{case}\, \alpha(\widetilde{v})\, \mathsf{in}\, \{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ t_i}$$

As $t = \mathsf{case}\, \alpha(\widetilde{v})\, \mathsf{in}\, \{\widetilde{T \mapsto t}\}$ and $\varphi \wr \emptyset \vdash \mathsf{case}\, \alpha(\widetilde{v})\, \mathsf{in}\, \{\widetilde{T \mapsto t}\}\colon T$, then by the type rules we know that the following holds:

$$\frac{\varphi_i \wr \emptyset \vdash t_i\colon T \qquad \epsilon \wr \emptyset \vdash v_i\colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \emptyset \vdash \mathsf{case}\, \alpha(\widetilde{v})\, \mathsf{in}\, \{\widetilde{T \mapsto t}\}\colon T}$$

By the reduction rule we know that $\exists T_i \in \widetilde{T}$ such that $\sigma(\alpha(\widetilde{v}))\colon T_i$. By the same reduction rule we know that $\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}$ reduces to the associated t_i. By the type rule we know that $\varphi_i \wr \emptyset \vdash t_i \colon T$, as required.

Consider the case where the last reduction rule used in the derivation is:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T' \qquad T' \notin \widetilde{T}}{[\sigma]\ \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}}$$

As $t = \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}$ and $\varphi \wr \emptyset \vdash \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}\colon T$, then by the type rules we know that the following holds:

$$\frac{\varphi_i \wr \emptyset \vdash t_i \colon T \qquad \epsilon \wr \emptyset \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \emptyset \vdash \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}\colon T}$$

Given that $\gamma = \alpha(\widetilde{v})$, by the definition of $\gamma' \vdash \gamma$ we know that then any relevant γ' must be $\alpha(\widetilde{T})$ where $\widetilde{v}\colon \widetilde{T}$. There are several possible derivations for the reduction $[\Lambda(\sigma)]\ \&(\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\gamma'}_F [\Lambda(\sigma'')]\ \varphi''$. it starts by one of the following rules.

$$\frac{\exists i.(\ [\Sigma]\ (\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \varphi_i' \wedge \texttt{error} \notin \varphi_i')}{[\Sigma]\ \&_I\,(\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \varphi_i'}$$

$$\frac{\forall i.(\ [\Sigma]\ (\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \varphi_i' \wedge \texttt{error} \in \varphi_i')}{[\Sigma]\ \&_I\,(\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \varphi_j'}$$

As in the hypotheses we assume that $\texttt{error} \notin \varphi''$ we can disregard the second rule. The derivation then proceeds by:

$$\frac{[\Sigma]\ (\alpha(\widetilde{T}), T_i) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \epsilon}{[\Sigma]\ (\alpha(\widetilde{T}), T_i);\ \varphi_i \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \varphi_i}$$

and finally by one of the following two rules:

$$\frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) = T_i}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T_i) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \epsilon} \qquad \frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) \neq T_i}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T_i) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \texttt{error}}$$

Consider when the first rule is used. Then we must have that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T_i$. However, by the reduction rule being considered we have that $\nexists i\,.\,\Lambda(\sigma)(\alpha(\widetilde{T})) = T_i$, as $\nexists i \,.\, \sigma(\alpha(\widetilde{v}))\colon T'' = T_i$, and by our assumptions we must have T'' be the same as $\Lambda(\sigma)(\alpha(\widetilde{v}))$. Hence we have a contradiction.

Consider when the second rule is used. Then we have that $[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \texttt{error}$. By our assumptions, however, we do not consider any such reductions. Hence we have a contradiction.

As our hypotheses are not fulfilled in this example, indeed they provide a contradiction, then we can disregard the $[\sigma]\ \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}$ reduction.

□

We can straightforwardly use Lemma 3.1 to prove a similar property for processes.

Lemma 3.2. *Thread Subject Reduction*

For all $\Phi, \Phi'', P, P', \sigma, \sigma'', \gamma, \gamma'$ if $\vdash P\colon \Phi$, and $[\sigma]\ P \xrightarrow{\gamma} [\sigma']\ P'$, and if whenever $\gamma' \vdash \gamma$ and $[\Lambda(\sigma)]\ \Phi \xrightarrow{\gamma'} [\Lambda(\sigma'')]\ \Phi''$ then $\texttt{error} \notin \varphi''$, then there exists a Φ' such that typing judgement $\vdash P'\colon \Phi'$ holds.

Proof. Suppose the hypotheses. We then prove the conclusions simultaneously by induction over the derivation of $[\sigma]\ P \xrightarrow{\gamma} [\sigma']\ P'$. □

3.4.1.2 Liveness

By Equation 3.9 we know that the effect of a Globally Compatible program point can either perform a reduction, or it consists of empty effects and recursion variables. Using this fact we can prove that such program points have various liveness properties. Firstly we can show that if an expression has an empty effect that it can either perform a reduction, or it is a value.

Lemma 3.3. *Empty Effect Expression Liveness*

For all t, T, if $\epsilon \wr \emptyset \vdash t\colon T$ then we know that either $t = v$, or there exists a t' such that $[\sigma]\ t \xrightarrow{\tau} [\sigma]\ t'$.

Proof. Suppose the hypotheses. We then prove the conclusions by induction over the derivation of $\epsilon \wr \emptyset \vdash t\colon T$. The key cases are as follows.

Consider the case where the last typing rule used in the derivation is:

$$\frac{\epsilon \wr \emptyset \vdash t_1\colon T_2 \xrightarrow{\epsilon} T_1 \qquad \epsilon \wr \emptyset \vdash t_2\colon T_2}{(\epsilon;\ \epsilon;\ \epsilon) \wr \emptyset \vdash t_1\, t_2\colon T_1}$$

As $\epsilon \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\epsilon} T_1$ and $\epsilon \wr \emptyset \vdash t_2 \colon T_2$ we can apply the inductive step to each of them and show that $[\sigma]\ t_1 \xrightarrow{\tau} [\sigma]\ t_1' \lor t_1 = v_1$, and $[\sigma]\ t_2 \xrightarrow{\tau} [\sigma]\ t_2' \lor t_2 = v_2$. If $[\sigma]\ t_1 \xrightarrow{\tau} [\sigma]\ t_1'$ then we can perform the following reduction:

$$\frac{[\sigma]\ t_1 \xrightarrow{\tau} [\sigma']\ t_1'}{[\sigma]\ t_1\, t_2 \xrightarrow{\tau} [\sigma']\ t_1'\, t_2}$$

as required. If $t_1 = v_1$ then we consider the properties of t_2. If $[\sigma]\ t_2 \xrightarrow{\tau} [\sigma]\ t_2'$ then, given that $t_1 = v_1$, we can perform the following reduction:

$$\frac{[\sigma]\ t_2 \xrightarrow{\tau} [\sigma']\ t_2'}{[\sigma]\ v\, t_2' \xrightarrow{\tau} [\sigma']\ v\, t_2'}$$

as required. The final case is when $t_1 = v_1$ and $t_2 = v_2$. By the typing rule, we know that $\epsilon \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\epsilon} T_1$, and hence that $t_1 = \texttt{rec}\, \underline{X}(x : T).t$. Hence we can perform the following reduction:

$$\frac{}{[\sigma]\ \texttt{rec}\, \underline{X}(x : T).t\, v_2 \xrightarrow{\tau} [\sigma]\ t[\texttt{rec}\, \underline{X}(x : T).t/\underline{X}][v_2/x]}$$

as required.

Consider the case where the last typing rule used in the derivation is:

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By the hypothesis we know that

$$(\alpha(\widetilde{v}), T) \equiv \epsilon$$

This is a contradiction, and hence we can disregard this case.

$\square$

We then consider non-empty effects. We show that if a program has an effect that can reduce then that program can always reduce.

Lemma 3.4. *Active Effect Expression Liveness*

For all $\varphi, \varphi', t, T, \gamma', \sigma', \sigma$ if $\varphi \wr \emptyset \vdash t \colon T$ and $[\Lambda(\sigma)]\ \varphi \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi'$ then there exists t', γ that $[\sigma]\ t \xrightarrow{\gamma} [\sigma']\ t'$.

Proof. Suppose the hypotheses. We then prove the conclusions by induction over the derivation of $\epsilon \wr \emptyset \vdash t \colon T$. The key cases are as follows.

Consider the case where the last typing rule used in the derivation is:

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By the hypothesis we know that $[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \rightarrow [\Lambda(\sigma')]\ \varphi'$. There are two relevant reduction rules for that derive this reduction:

$$\frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) = T}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \epsilon} \qquad \frac{\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \quad \Lambda(\sigma)(\alpha(\widetilde{T})) \neq T}{[\Lambda(\sigma)]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Lambda(\sigma')]\ \texttt{error}}$$

Consider if the reduction is performed using the former. By our assumptions about the abstract and concrete state transition functions we know that the abstract state can reduce if and only if the concrete state can reduce, on a given action. By the premise of the reduction we know that $\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma')$, and hence that $\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$. We also know that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T$. By our assumptions about the abstract and concrete state projection function we know that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T$ if and only if $\sigma(\alpha(\widetilde{v})) \colon T$. We then have all the premises required in order to use the following rule:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T}{[\sigma]\ (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \sigma(\alpha(\widetilde{v}))}$$

to reduce $(\alpha(\widetilde{v}), T)$ as required.

Consider if the reduction is performed using the latter. By our assumptions about the abstract and concrete state transition functions we know that the abstract state can reduce if and only if the concrete state can reduce, on a given action. By the premise of the reduction we know that $\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma')$, and hence that $\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$. We also know that $\Lambda(\sigma)(\alpha(\widetilde{T})) \neq T$. By our assumptions about the abstract and concrete state projection function we know that $\Lambda(\sigma)(\alpha(\widetilde{T})) = T$ if and only if $\sigma(\alpha(\widetilde{v})) \colon T'$ and $T' \neq T$. We then have all the premises required in order to use the following rule:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T' \neq T}{[\sigma]\ (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\ \texttt{error}}$$

to reduce $(\alpha(\widetilde{v}), T)$ as required.

Consider the case where the last typing rule used in the derivation is:

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \emptyset \vdash t_2 \colon T_2}{(\varphi_1;\ \varphi_2;\ \varphi_3) \wr \emptyset \vdash t_1\, t_2 \colon T_1}$$

By the hypotheses we know that $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi'$, and by using a combination of the rule for sequence and equivalent effect reduction:

$$\frac{[\Sigma]\ \varphi_1 \xrightarrow{\gamma} [\Sigma']\ \varphi_1'}{[\Sigma]\ \varphi_1;\,\varphi_2 \xrightarrow{\gamma} [\Sigma']\ \varphi_1';\,\varphi_2} \qquad \frac{\Phi \equiv \Phi_1 \quad [\Sigma]\ \Phi_1 \xrightarrow{\gamma} [\Sigma']\ \Phi_2 \quad \Phi_2 \equiv \Phi'}{[\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi'}$$

we know that either $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_1';\,\varphi_2;\,\varphi_3$ or $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_2';\,\varphi_3$ where $\varphi_1 \equiv \epsilon$, or $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_3'$ where $\varphi_1 \equiv \varphi_2 \equiv \epsilon$. In each case we also know that $[\Lambda(\sigma)]\ \varphi_i \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_i'$.

Consider the case where $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_1';\,\varphi_2;\,\varphi_3$. As $\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1$ and $[\Lambda(\sigma)]\ \varphi_1 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_1'$ we can apply the inductive step and hence know that $[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'$. Hence we can reduce $t_1\,t_2$ using the following rule:

$$\frac{[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'}{[\sigma]\ t_1\,t_2 \xrightarrow{\gamma} [\sigma']\ t_1'\,t_2}$$

as required.

Consider the case where $[\Lambda(\sigma)]\ \varphi_1;\,\varphi_2;\,\varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_2';\,\varphi_3$. As $\varphi_2 \wr \emptyset \vdash t_2 \colon T_2$ and $[\Lambda(\sigma)]\ \varphi_2 \xrightarrow{\gamma'} [\Lambda(\sigma')]\ \varphi_2'$ we can apply the inductive step and hence know that $[\sigma]\ t_2 \xrightarrow{\gamma} [\sigma']\ t_2'$. By Lemma 3.3 we know that, as $\varphi_1 \equiv \epsilon$ that either $[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'$ or $t_1 = v_1$. If the former then we can reduce $t_1\,t_2$ using the following rule:

$$\frac{[\sigma]\ t_1 \xrightarrow{\gamma} [\sigma']\ t_1'}{[\sigma]\ t_1\,t_2 \xrightarrow{\gamma} [\sigma']\ t_1'\,t_2}$$

as required. If the latter then we can reduce $t_1\,t_2$ using the following rule:

$$\frac{[\sigma]\ t_2 \xrightarrow{\gamma} [\sigma']\ t_2'}{[\sigma]\ v\,t_2' \xrightarrow{\gamma} [\sigma']\ v\,t_2'}$$

as required. Consider the case when $t_1 = v_1$. By Lemma 3.3 we know that, as $\varphi_2 \equiv \epsilon$ that either $[\sigma]\ t_2 \xrightarrow{\gamma} [\sigma']\ t_2'$ or $t_2 = v_2$. If the former then we can reduce $t_1\,t_2$ using the following rule:

$$\frac{[\sigma]\ t_2 \xrightarrow{\gamma} [\sigma']\ t_2'}{[\sigma]\ v\,t_2' \xrightarrow{\gamma} [\sigma']\ v\,t_2'}$$

as required. If $t_1 = v_1$ and $t_2 = v_2$ then, as $\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1$, we know that $t_1 = \mathtt{rec}\,\underline{X}(x : T).t$, and hence can reduce $t_1\,t_2$ using the following rule:

$$\frac{}{[\sigma]\ \mathtt{rec}\,\underline{X}(x : T).t\,v \xrightarrow{\tau} [\sigma]\ t[\mathtt{rec}\,\underline{X}(x : T).t/\underline{X}][v/x]}$$

as required.

The case where $\boxed{[\Lambda(\sigma)]}\ \varphi_1;\ \varphi_2;\ \varphi_3 \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma')]}\ \varphi_3'$ is similar to the case where $\boxed{[\Lambda(\sigma)]}\ \varphi_1;\ \varphi_2;\ \varphi_3 \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma')]}\ \varphi_2';\ \varphi_3$.

□

We can extend these results straightforwardly to threads.

Lemma 3.5. *Empty Effect Thread Liveness Lemma*

For all P*, if* $\vdash P\colon \Pi_I\, \epsilon$ *then we know that either* $P = \Pi_I\, v$*, or there exists a* P' *such that* $\boxed{[\sigma]}\ P \xrightarrow{\tau} \boxed{[\sigma]}\ P'$*.*

Proof. Suppose the hypotheses. We then simultaneously prove the conclusions by induction over the derivation of $\vdash P\colon \Pi_I\, \epsilon$. □

Lemma 3.6. *Active Effect Thread Liveness Lemma*

For all $\Phi, \Phi', P, \gamma', \sigma', \sigma$ *if* $\vdash P\colon \Phi$ *and* $\boxed{[\Lambda(\sigma)]}\ \Phi \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma')]}\ \Phi'$ *then there exists* P', γ *that* $\boxed{[\sigma]}\ P \xrightarrow{\gamma} \boxed{[\sigma']}\ P'$*.*

Proof. Suppose the hypotheses. We then prove the conclusions by induction over the derivation of $\vdash P\colon \Phi$. □

3.4.2 Key Theorems

Checking whether Global Compatibility holds for a given system amounts to model checking, as the predicate involves considering all possible interleavings of effect reductions. As such a check is computationally expensive we permit an analysis to use stronger properties that imply Global Compatibility, which are often easier to prove. We express this using the following type rule:

$$\frac{\vdash P\colon \Phi \qquad \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma), \Phi) \text{ and } (\mathcal{C} \implies \mathcal{G})}{\vdash \boxed{[\sigma]}\ P} \tag{3.12}$$

This denotes that, if we can find a property which implies Global Compatibility, then we consider programs and states that fulfil that property to be Globally Compatible. We refer to program points which can be validated using the rule in Equation 3.12 to be *valid*. We explore the stronger properties in Section 3.5.

We can prove key theorems using the lemmas in Section 3.4.1. Firstly we prove Subject Reduction. This states that if a valid program point performs a reduction, that the new program point is also valid.

Theorem 3.7. *Valid Program Points Subject Reduction*

$$\vdash \boxed{[\sigma]}\ P \wedge \boxed{[\sigma]}\ P \rightarrow \boxed{[\sigma']}\ P' \quad \Rightarrow \quad \vdash \boxed{[\sigma']}\ P'$$

Proof.

We know that:
$$\frac{\vdash P : \Phi \qquad \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma), \Phi) \text{ and } (\mathcal{C} \implies \mathcal{G})}{\vdash \boxed{[\sigma]}\ P}$$

By the definition of global compatibility we know that Equation 3.8 holds. We can then show that:

$$\boxed{[\Sigma]}\ \varphi \rightarrow \boxed{[\Sigma']}\ \varphi' \Rightarrow \nexists \texttt{error} \in \varphi' \tag{3.13}$$

Hence we can use Lemma 3.2 to show, by choosing $\mathcal{C} = \mathcal{G}$ that:

$$\vdash P' : \Phi' \qquad \boxed{[\Lambda(\sigma)]}\ \Phi \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma')]}\ \Phi' \tag{3.14}$$

By Equation 3.8 we know that $\mathcal{G}(\Sigma', \Phi')$ holds. Hence we can show that:

$$\frac{\vdash P' : \Phi' \qquad \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma'), \Phi') \text{ and } (\mathcal{C} \implies \mathcal{G})}{\vdash \boxed{[\sigma']}\ P'}$$

□

We also prove that all valid program points are live; this denotes that a set of threads can either reduce, or each thread consists of a value. In particular this denotes, if there are several possible actions in reducible positions, that at least one must not be blocked.

Theorem 3.8. *Liveness of Valid Program Points*

$$\vdash \boxed{[\sigma]}\ P \Rightarrow \boxed{[\sigma]}\ P \rightarrow \boxed{[\sigma']}\ P' \vee P = \Pi\, v$$

Proof. By the definition of global compatibility we know that Equation 3.9 holds. We can perform a case analysis on the two disjuncts.

$$\boxed{[\Sigma]}\ \Phi \rightarrow \boxed{[\Sigma']}\ \Phi \vee \Phi \equiv \Pi\, \epsilon \tag{3.15}$$

Case $(\Phi = \Pi\, \epsilon)$

By Lemma 3.5 we have that:

$$\boxed{[\sigma]}\ P \xrightarrow{\gamma} \boxed{[\sigma]}\ P' \vee P = \Pi_I\, v \tag{3.16}$$

Case $\boxed{[\Sigma]}\ \Phi \rightarrow \boxed{[\Sigma']}\ \Phi'$

$$\begin{array}{lcll}
d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_M \restriction d_a & \overset{\text{def}}{=} & \oplus_N(c!\langle T_n\rangle;\, G_n \restriction d_a) & \emptyset \neq N \subseteq M \\
d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_M \restriction d_b & \overset{\text{def}}{=} & \&_M(c?(T_m);\, G_m \restriction d_b) & \\
d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_M \restriction d & \overset{\text{def}}{=} & G_1 \restriction d & \\
(\mu \underline{X}.G') \restriction d & \overset{\text{def}}{=} & \mu \underline{X}.(G' \restriction d) & \\
\underline{X} \restriction d & \overset{\text{def}}{=} & \underline{X} & \\
\mathsf{0} \restriction d & \overset{\text{def}}{=} & \mathsf{0} &
\end{array}$$

Figure 3.12: Global Session Types Projection

By Lemma 3.6.

$$[\sigma]\ P \xrightarrow{\gamma} [\sigma']\ P' \tag{3.17}$$

□

Theorems 3.7 and 3.8 can also be proved as corollaries of Theorems 4.1 and 4.2 respectively

3.5 Specific Formulations

Instead of investigating all possible inter leavings of a system, we attempt to prove stronger properties that imply Global Compatibility, as these properties are often easier to prove. The challenge is then to define a $\mathcal{C}$ such that $\mathcal{C} \implies \mathcal{G}$. In this section we describe two such compatibility predicates, one for message passing systems that use blocking receives (Section 3.5.1), and one for message passing systems that use non-blocking receives (Section 3.5.2).

3.5.1 Blocking Message Passing

Global Session Types (Figure 2.9) describe communication protocols between multiple role at a high level of abstraction (Bettini et al., 2008). Intuitively each participant role is played out by one thread in a program. We make use of the projection of Global Session Types to effects (Figure 3.12). Note that, in the case where the participant being projected to is neither the sender nor the receiver, we require that the projection of all the continuations be the same, so we (arbitrarily) project for the first one. We use the notation $c!\langle T\rangle$ as shorthand for $(\texttt{snd}(\mathsf{Res}\, c, T), \textsc{Unit})$ and $c?(T)$ as shorthand for $(\texttt{rcv}(\mathsf{Res}\, c), T)$.

In order for a program to avoid communication errors and deadlock it is not sufficient for the effect of the program to be a projection of a Global Session Type. Consider the

$$\Delta \overset{\text{def}}{=} \bigcup_{n<\omega} \Delta_n \qquad \Delta_0 \overset{\text{def}}{=} \{\, [\Sigma_\emptyset]\ \Phi^{\upharpoonright G} \}$$

$$\begin{aligned}
\Delta_{n+1} \overset{\text{def}}{=} \quad & \{\, \delta(\,[c \mapsto T;\, \Sigma^k]\,, \&_J(c?(T_j);\, \varphi_b^j), \varphi_a^k) \mid \forall j \in J. \delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n)\} \\
& \cup \{\, \delta(\,[\Sigma]\,, \oplus_K(c!\langle T_k\rangle;\ \varphi_a^k), \&_J(c?(T_j);\ \varphi_b^j) \mid \\
& \qquad \forall k \in K \,.\, \delta(\Sigma, \varphi_a^k, \varphi_b^k) \in \Delta_n \wedge \Sigma(c) = \emptyset) \\
& \cup \Delta_n
\end{aligned}$$

where $c = d_a d_b$ and $\emptyset \subset K \subseteq J$.

Figure 3.13: Blocking Message Passing Safe Program Points Set

following program:

$$\begin{array}{ll}
1 & [c \mapsto 4]\ \texttt{snd}(c, \texttt{true}).0 \parallel \texttt{rcv}(c)(x : \textsc{Bool}).0 \\
2 & [\rightarrow]\ \ [c \mapsto 4, \texttt{true}]\ 0 \parallel \texttt{rcv}(c)(x : \textsc{Bool}).0 \\
3 & [\rightarrow]\ \ [c \mapsto \texttt{true}]\ 0 \parallel \texttt{error}
\end{array} \tag{3.18}$$

The effect of the code on line 1 is $c!\langle\textsc{Bool}\rangle \parallel c?(\textsc{Bool})$ which is complementary, according to the standard definition (Honda, 1993). However, since there are messages in the queue, in order to rule out communication errors and deadlock we cannot consider solely the effect of the code. We must also consider the state of the message queues. We do this by ensuring that the messages in the queue are complementary to the effect of the code, and that after that point the effects are complementary (Bettini et al., 2008).

3.5.1.1 Safe Program Points

We use the notation $\delta(\Sigma, \varphi_i, \varphi_j)$ to represent some parallel configuration with two particular participants p_i, p_j identified:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi_j \parallel \ldots \parallel \varphi_n$$

By writing $\delta(\Sigma', \varphi_i', \varphi_j')$ we refer to this same configuration but with possible changes in the abstract state and identified participants' effects. We make use of $\delta(\Sigma, \varphi_i)$ to similarly single out one participant from a parallel configuration.

We inductively define a set of safe program points, which can be reduced without causing error or deadlock (Figure 3.13). The base case Δ_0 consists of the projection of global session types, alongside empty message queues. The inductive step defines new valid points, using existing valid points, in one of two ways. Intuitively, the first adds a message to the front of the queue and a receive action that can expects a value of that type. The second adds a send and a receive that match up in terms of the type of values sent and received. The above intuition is complicated by the provision of choice, as is explained below. We explain these cases by example.

Adding a message. Consider the following Global Session Type, and a program point of its projection:

$$\begin{aligned} G \quad = \quad & d_1 \to d_2 \colon c\ \langle l_1 \mapsto \\ & \qquad d_1 \to d_2 \colon c\ \langle T_1 \mapsto \mathsf{0} \rangle \\ & \rangle \end{aligned} \tag{3.19}$$

$$\boxed{[\Sigma_\emptyset]}\ c!\langle l_1 \rangle;\ c!\langle T_1 \rangle \parallel c?(l_1);\ c?(T_1) \in \Delta_0 \tag{3.20}$$

This program point can perform a reduction that sends the label on the shared channel:

$$\boxed{[\Sigma_\emptyset]}\ c!\langle l_1 \rangle;\ c!\langle T_1 \rangle \parallel c?(l_1);\ c?(T_1) \to \boxed{[c \mapsto l_1]}\ c!\langle T_1 \rangle \parallel c?(l_1);\ c?(T_1) \tag{3.21}$$

We can show that this new program point, derived from the first, is in the set of safe program points, using the extensions defined in Figure 3.13. Consider the global session type $d_1 \to d_2 :\ c\,\langle T_1 \mapsto 0 \rangle$. We can define a program point that is a projection of this global session type, and hence in Δ_0:

$$\boxed{[\Sigma_\emptyset]}\ c!\langle T_1 \rangle \parallel c?(T_1) \tag{3.22}$$

This point can be extended by adding a message of the correct type to the queue, and adding a receive action of the same type, to the receiving process:

$$\boxed{[c \mapsto \mathtt{Lab}\, l_1]}\ c!\langle T_1 \rangle \parallel c?(\mathtt{Lab}\, l_1);\ c?(T_1) \tag{3.23}$$

This approach of adding a message onto a channel queue, and a corresponding receive, is relatively straightforward, and forms the basis of the first inductive definition in Figure 3.13. Were this the limit of what we wished to permit, we could define this first inductive step as:

$$\{\, \delta(\boxed{[c \mapsto T;\ \Sigma]}\,, c?(T_j); \varphi_b,\ \varphi_a) \mid \delta(\Sigma, \varphi_b,\ \varphi_a) \in \Delta_n)\} \tag{3.24}$$

When a value is received, however, it is possible for a case split to be performed on its type. In order to account for this case we must construct the set in a slightly more complex manner. Consider the following global session type:

$$\begin{aligned} G \quad = \quad & d_1 \to d_2 \colon c\langle \\ & \qquad l_1 \mapsto (d_1 \to d_2 \colon c\langle T_1 \mapsto \mathsf{0} \rangle) \\ & \qquad l_2 \mapsto (d_1 \to d_2 \colon c\langle T_2 \mapsto \mathsf{0} \rangle) \\ & \rangle \end{aligned} \tag{3.25}$$

and a program point with one of its projections:

$$\boxed{[\Sigma_\emptyset]}\ c!\langle \mathtt{Lab}\, l_1 \rangle;\ c!\langle T_1 \rangle \parallel (c?(\mathtt{Lab}\, l_1);\ c?(T_1))\&(c?(\mathtt{Lab}\, l_2);\ c?(T_2)) \in \Delta_0 \tag{3.26}$$

This program point can perform a reduction that sends the label on the shared channel:

$$\begin{aligned}\boxed{[\Sigma_\emptyset]}\ c!\langle \texttt{Lab}\, l_1\rangle;\ c!\langle T_1\rangle \parallel (c?(\texttt{Lab}\, l_1);\ c?(T_1))\&(c?(\texttt{Lab}\, l_2);\ c?(T_2)) \rightarrow \\ \boxed{[c \mapsto \texttt{Lab}\, l_1]}\ c!\langle T_1\rangle \parallel (c?(\texttt{Lab}\, l_1);\ c?(T_1))\&(c?(\texttt{Lab}\, l_2);\ c?(T_2))\end{aligned} \tag{3.27}$$

Using the former approach we can straightforwardly show that the following program points are valid:

$$\boxed{[c \mapsto \texttt{Lab}\, l_1]}\ c!\langle T_1\rangle \parallel c?(\texttt{Lab}\, l_1);\ c?(T_1) \qquad \boxed{[c \mapsto \texttt{Lab}\, l_2]}\ c!\langle T_2\rangle \parallel c?(\texttt{Lab}\, l_2);\ c?(T_2) \tag{3.28}$$

We can combine these two points, using the state and the effect of the sender from one program point, and performing an external choice between the effects of the receiver in each program point, to generate the program point:

$$\boxed{[c \mapsto \texttt{Lab}\, l_1]}\ c!\langle T_1\rangle \parallel (c?(\texttt{Lab}\, l_1);\ c?(T_1))\&(c?(\texttt{Lab}\, l_2);\ c?(T_2)) \tag{3.29}$$

Note that the two points being combined have different accompanying states, and this is reflected in the precondition for this construction $\delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n$.

In the full version of the 'adding a message' case, we take a set of valid program points $\{\varphi_{j_1}, \ldots, \varphi_{j_n}\}$, such as those in Equation 3.28, where the effects of all threads, apart from those of the proposed sender and receiver, are the same. We sequence a receive action of the associated type onto each continuation, for the receive effects, so that we obtain $\{c?(T_{j_1});\ \varphi_b^{j_1}, \ldots, c?(T_{j_n});\ \varphi_b^{j_n}\}$. We define an external choice between these effects, and obtain

$$c?(T_{j_1});\ \varphi_b^{j_1} \& \ldots \& \, c?(T_{j_n});\ \varphi_b^{j_n} \tag{3.30}$$

We then choose some k in $j_1 \ldots j_n$, and add a message T_k to the front of the queue of the designated channel between the sender and the receiver. As the $T_{j_1}, \ldots, T_{j_n}$ are required to be distinct, we know that $c?(T_{j_1});\ \varphi_b^{j_1} \& \ldots \& \, c?(T_{j_n});\ \varphi_b^{j_n}$, with message T_k at the front of its queue, can only ever evolve to φ_b^k. Therefore, in order to maintain duality, we pair this external choice with the effect φ_a^k, obtaining a final definition:

$$\{\, \delta(\ \boxed{[c \mapsto T;\ \Sigma^k]}\ , \&_J(c?(T_j); \varphi_b^j), \varphi_a^k) \mid \forall j \in J. \delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n)\} \tag{3.31}$$

Adding a send-receive pair. Consider the following Global Session Type:

$$d_1 \rightarrow d_2 :\ c_1(T_1); d_3 \rightarrow d_2 :\ c_2(T_3) \tag{3.32}$$

whose projection is:

$$\boxed{[\Sigma_\emptyset]}\ c_1!\langle T_1\rangle \,||\, c_1?(T_1);\ c_2?(T_3) \,||\, c_2!\langle T_3\rangle \in \Delta_0 \tag{3.33}$$

We can show that we can derive this program point from the projection of the session:

$$d_3 \to d_2 : c_2(T_3) \tag{3.34}$$

whose projection is:

$$\boxed{[\Sigma_\emptyset]}\ \epsilon \,||\, c_2?(T_3) \,||\, c_2!\langle T_3\rangle \in \Delta_0 \tag{3.35}$$

This point can be extended by adding the dual pair of $c_1!\langle T_1\rangle$ and $c_1?(T_1)$ between the first and second processes

$$\boxed{[\Sigma_\emptyset]}\ c_1!\langle T_1\rangle \,||\, c_1?(T_1);\, c_2?(T_3) \,||\, c_2!\langle T_3\rangle \tag{3.36}$$

which returns us to the original projection. Given that the original was also a projection of a global session type, this may seem like a redundant construction, but it is essential when we consider how the $c_2?(T_3) \,||\, c_2!\langle T_3\rangle$ interaction can occur with respect to the $c_1!\langle T_1\rangle \,||\, c_1?(T_1)$ interaction.

We can show that the full system can reduce as:

$$\boxed{[\Sigma_\emptyset]}\ c_1!\langle T_1\rangle \,||\, c_1?(T_1);\, c_2?(T_3) \,||\, c_2!\langle T_3\rangle \xrightarrow{c_2!\langle T_3\rangle} \boxed{[c_2 \mapsto T_3]}\ c_1!\langle T_1\rangle \,||\, c_1?(T_1);\, c_2?(T_3) \,||\, \epsilon \tag{3.37}$$

This particular program point cannot be constructed by projecting a global session type and then using the emphadding a message technique, as that adds message/receive pairs to the *front* of the effect of the receiving process. Instead, we start with an empty session, and directly add a message/receive pair between the third and second processes:

$$\boxed{[c_2 \mapsto T_3]}\ \epsilon \,||\, c_2?(T_3) \,||\, \epsilon \tag{3.38}$$

We then make use of the *adding a send/receive pair* to add the not-yet-commenced dual interaction between the first and the second processes

$$\boxed{[c_2 \mapsto T_3]}\ c_1!\langle T_1\rangle \,||\, c_1?(T_1);\, c_2?(T_3) \,||\, \epsilon \tag{3.39}$$

Note the side condition for this case, that the channel queue being added to must be empty. If we didn't have this side condition, then we could construct a program point such as

$$\boxed{[c_2 \mapsto T_3]}\ \epsilon \,||\, c_2?(T_1);\, c_2?(T_3) \,||\, c_2!\langle T_1\rangle;\epsilon \tag{3.40}$$

by adding the send/receive pair $c_2!\langle T_1\rangle$ and $c_2?(T_1)$ to the third and second processes respectively. As the message queue for c_2 is not empty, and we can place a receive expecting an arbitrary type at the front of the receiving process, the existing message in the queue will probably not be compatible with the first receive action of the receiving thread.

This construction makes use of a key design decision. We use a separate channel for each direction of each pair of communicating roles (principle channel allocation (Deniélou and Yoshida, 2010)). This enables us to determine that it is safe to add a send/receive pair in front of existing message/receive pairs. The requirement that the channel being used is empty, and that the receiving process can only receive from the sending process on the channel being used, guarantees that the send/receive pair being added will not interact in error with any existing messages. For all message passing instances, for the remainder of the thesis, we assume that principle channel allocation is used.

3.5.1.2 Formal Properties

We can prove that if a program point is in the valid points set for a given G, and if the point reduces, then it stays in a valid points set, for a possibly different G'.

Lemma 3.9. *Valid Points Set Subject Reduction*

$$\forall \Sigma, \Phi, n, \Sigma', \Phi'.\ [\Sigma]\ \Phi \in \Delta_n \wedge [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow [\Sigma']\ \Phi' \in \Delta_{n+1}$$

Proof. Suppose the hypotheses. We then prove the conclusion by induction over the n of Δ_n.

Consider the case where $n = 0$. We then continue by a case analysis on the last reduction rule used in the derivation of $[\Sigma]\ \Phi \to [\Sigma']\ \Phi'$.

Consider the case when the last reduction performed is an internal choice. By the definition of Δ_n in Figure 3.13, we know that $[\Sigma]\ \Phi$ is of the form:

$$[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel \oplus_N c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m$$

Hence,by the rule for internal choice:

$$\frac{i \in 1, 2}{[\Sigma]\ \varphi_1 \oplus \varphi_2 \xrightarrow{\tau} [\Sigma]\ \varphi_i}$$

we know that:

$$[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel \oplus_N c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m \to [\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m$$

By Lemma B.2 we know that: $\exists G' = d_a \to d_b \colon c\langle \widetilde{T \mapsto G}\rangle_M . N \subseteq M \wedge \Pi\, G' \upharpoonright d_i \equiv \Pi\, \varphi_i$ By the definition of the projection relation $\upharpoonright$ we know that it is possible to project G' such that $G' \upharpoonright d_a \equiv c!\langle T_n\rangle;\ \varphi_a^n$. Hence we can show that $\varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m \equiv \Pi\, G' \upharpoonright d_i$ and $[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m \in \Delta_0$, as required.

Consider the case when the last reduction performed is a send action, in sequence with some continuation. By the definition of Δ_n in Figure 3.13, we know that $[\Sigma]\ \Phi$ is of the form $[\Sigma]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m$. By the action reduction and sequencing rules:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \epsilon} \qquad \frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) \neq T}{[\Sigma]\ (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\ \texttt{error}}$$

$$\frac{[\Sigma]\ \varphi_1 \xrightarrow{\gamma}_F [\Sigma']\ \varphi_1'}{[\Sigma]\ \varphi_1;\ \varphi_2 \xrightarrow{\gamma}_F [\Sigma']\ \varphi_1';\ \varphi_2}$$

we know that, if $[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_o \to [c \mapsto T_n]\ \varphi_1 \parallel \ldots \parallel \varphi_a^n \parallel \ldots \parallel \varphi_o$ then $[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n\rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_o \to [c \mapsto T_n]\ \varphi_1 \parallel \ldots \parallel \varphi_a^n \parallel \ldots \parallel \varphi_o$.

By the first part of the definition of Δ_{n+1} in Figure 3.13, in order to show that $\delta(c \mapsto T, \varphi_a^k, \&_J(c?(T_j);\ \varphi_b^j)) \in \Delta_1$, we need to show that for each $j \in J$ that $\delta(\Sigma_\emptyset, \varphi_a^k, \varphi_b^j)) \in \Delta_0$.

By Lemma B.2 we know that $\exists G' = d_a \to d_b\colon c\langle\widetilde{T \mapsto G}\rangle_M \,.\, \Pi\, G' \restriction d_i \equiv \Pi\, \varphi_i$. Hence we have that, for each of the threads that are not the sender or receiver threads, that $\varphi_i \equiv G' \restriction d_i \equiv G_d \restriction d_i$, for any $d \in M$. In such cases let $\varphi_i' \equiv \varphi_i$. By the definition of projection we know that $G' \restriction d_b \equiv \&_M c?(T_m);\ (G_m \restriction d_b) \equiv \&_M c?(T_m);\ {\varphi'}_b^{m}$ and $G' \restriction d_a \equiv c!\langle T_n\rangle;\ (G_n \restriction d_a) \equiv c!\langle T_n\rangle;\ {\varphi'}_a^{n}$ Hence we have that $[\Sigma_\emptyset]\ \Pi\, {\varphi'}_i^{m} \equiv \Pi\, G_m \restriction d_i \in \Delta_0$, as required.

Consider the case where $n > 0$. By the structure of parallel effects we can identify one effect which is the one that reduces:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi_m \to [\Sigma']\ \varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \varphi_m$$

As $[\Sigma]\ \Phi \in \Delta_n$ we know that either in one of the constructs included in the definition of Δ_n, or it is in Δ_{n-1}.

Consider the case when $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \&_J(c?(T_j);\ \varphi_b^j))\}$ where $\forall j \in J. \delta(\Sigma^j, \varphi_b^j) \in \Delta_{n-1}$ (as in Figure 3.13). We can perform a case split over the index i. If $i = b$ then, by the reduction rules, we know that $\varphi_i' \equiv \varphi_b^k$ and that $\Sigma' = \Sigma_k$. Hence we have that $[\Sigma']\ \Phi' \equiv \delta(\Sigma^k, \varphi_a^k, \varphi_b^k)$. Hence by our hypotheses we know that $[\Sigma']\ \Phi' \in \Delta_n$. Consider the case when $i \neq b$. By Lemma B.1 we know that:

$$[c \mapsto T; \Sigma^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi_m \to [c \mapsto T; \Sigma'^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \varphi_m$$

if and only if:

$$\begin{aligned}[\Sigma^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \&_J(c?(T_j);\ \varphi_b^j) \parallel \ldots \parallel \varphi_m \to\\ [\Sigma'^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \&_J(c?(T_j);\ \varphi_b^j) \parallel \ldots \parallel \varphi_m\end{aligned}$$

Then, by the parallel reduction rules, we know that:

$$[\Sigma^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi^k_b \parallel \ldots \parallel \varphi_m \rightarrow [\Sigma'^k]\ \varphi_1 \parallel \ldots \parallel \varphi'_i \parallel \ldots \parallel \varphi^k_b \parallel \ldots \parallel \varphi_m$$

By the hypotheses we know that $[\Sigma^k]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi^k_b \parallel \ldots \parallel \varphi_m \in \Delta_{n-1}$. Hence, by the inductive step, we know that $[\Sigma'^k]\ \varphi_1 \parallel \ldots \parallel \varphi'_i \parallel \ldots \parallel \varphi^k_b \parallel \ldots \parallel \varphi_m \in \Delta_{n-1}$. By the definition of $\Delta^{\restriction G}_n$ in Figure 3.13 we hence know that:

$$[\Sigma'^k]\ \varphi_1 \parallel \ldots \parallel \varphi'_i \parallel \ldots \parallel \&_J(c?(T_j);\ \varphi^j_b) \parallel \ldots \parallel \varphi_m \in \Delta n$$

as required.

Consider the case when $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \oplus_K(c!\langle T_k\rangle;\ \varphi^k_a), \&_J(c?(T_j);\ \varphi^j_b))\}$ where $\forall j \in J.\delta(\Sigma^j, \varphi^j_a, \varphi^j_b) \in \Delta^{\restriction G}_{n-1}$ and $\Sigma(c) = \emptyset$ (as in Figure 3.13). We can perform a case split over the index i. If $i = b$ then, by the reduction rules we know that $\Sigma(c) = T, q$, which contradicts the hypotheses, and hence can disregard this case. Consider the case when $i = a$. If $\oplus_K(c!\langle T_k\rangle;\ \varphi^k_a)$ is a choice then we can use the internal choice reduction rules to show that:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \oplus_K(c!\langle T_k\rangle;\ \varphi^k_a) \parallel \ldots \parallel \varphi_m \rightarrow [\Sigma]\ \varphi_1 \parallel \ldots \parallel \oplus_L(c!\langle T_l\rangle;\ \varphi^l_a) \parallel \ldots \parallel \varphi_m$$

for some L such that $\emptyset \subset L \subseteq J$. We straightforwardly have that $\varphi_1 \parallel \ldots \parallel \oplus_L(c!\langle T_l\rangle;\ \varphi^l_a) \parallel \ldots \parallel \varphi_m \in \Delta_n$. If $\oplus_K(c!\langle T_k\rangle;\ \varphi^k_a)$ is a vacuous choice then it is simply of the form $c!\langle T_k\rangle;\ \varphi^k_a$. By the reduction rules then we know that $\Sigma' = \Sigma_k;\ c \mapsto T_k$ and $\varphi'_a = \varphi^k_a$. By the hypotheses we know that $\delta(\Sigma^k, \varphi^k_a, \varphi^k_b) \in \Delta_{n-1}$. Then, by the first part of the definition of $\Delta^{\restriction G}_n$ in Figure 3.13 we know that $[\Sigma']\ \Phi' = \delta([c \mapsto T; \Sigma^k], \varphi^k_a, \&_J(c?(T_j);\ \varphi^j_b)) \in \Delta_n$ as required. Consider the case when $i \neq b$ and $i \neq a$. This follows similarly to the case where $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \varphi^k_a, \&_J(c?(T_j);\ \varphi^j_b))\}$ and $i \neq b$, above.

Consider the case when $[\Sigma]\ \Phi \in \Delta_{n-1}$. By the inductive hypothesis we have that $[\Sigma']\ \Phi' \in \Delta_{n-1}$, and hence that $[\Sigma']\ \Phi' \in \Delta_n$ as required. $\square$

We can also prove relatively straightforwardly that if program point is in one of the Δ_n of Δ then it contains no errors, and that it is live or empty.

Lemma 3.10. *Valid States Safety Lemma*

$$\forall \Sigma, \Phi, n.\ [\Sigma]\ \Phi \in \Delta_n \Rightarrow \mathtt{error} \notin \Phi$$

Proof. Suppose the hypothesis. We then prove the conclusion by induction over n. $\square$

Lemma 3.11. *Valid States Liveness*

$$\forall \Sigma, \Phi, n.\ [\Sigma]\ \Phi \in \Delta_n \Rightarrow \exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \vee \Phi \equiv \Pi\, \epsilon$$

Proof. Suppose the hypothesis. We then prove that one of the two conclusions hold, by induction over the n of Δ_n.

Consider the case where $n = 0$. We then proceed by case analysis of G.

If $[\Sigma_\emptyset]\ \Phi \in \Delta_0$ and $\Phi \equiv \Phi'$ then $[\Sigma_\emptyset]\ \Phi' \in \Delta_0$ and hence it is sufficient to consider direct reduction.

Consider the case where $G = d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle$. The global session type can be projected in different ways for the d_a component of the effect. If $d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle \restriction d_a \equiv \oplus_N (c!\langle T_n \rangle;\ G_n \restriction d_a)$ then by the reduction rules we know that we can perform the following reduction:

$$[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel \oplus_N (c!\langle T_n \rangle;\ G_n \restriction d_a) \parallel \ldots \parallel \varphi_o \to [\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel (c!\langle T_n \rangle;\ G_n \restriction d_a) \parallel \ldots \parallel \varphi_o$$

If, however, the global session type is projected to a send followed by a continuation, rather than a choice between such effects, such as $d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle \restriction d_a \equiv \varphi_1 \parallel \ldots \parallel (c!\langle T_n \rangle;\ G_n \restriction d_a) \parallel \ldots \parallel \varphi_o$, then we know that the effect can perform the following reduction:

$$[\Sigma_\emptyset]\ \varphi_1 \parallel \ldots \parallel (c!\langle T_n \rangle;\ G_n \restriction d_a) \parallel \ldots \parallel \varphi_o \to [c \mapsto T_n]\ \varphi_1 \parallel \ldots \parallel G_n \restriction d_a \parallel \ldots \parallel \varphi_o$$

Hence we have that $[\Sigma_\emptyset]\ \Phi$ is live, as required.

Consider the case where $G = \mathsf{0}$. Here we have that $\mathsf{0} \restriction d_i \equiv \epsilon$ for each d_i. Hence $\Phi \equiv \Pi\, \epsilon$, as required.

Consider the case where $n > 0$. As $[\Sigma]\ \Phi \in \Delta_n$ then by Figure 3.13 we know that $[\Sigma]\ \Phi$ is of one of the following forms.

If it is of the form:

$$\{\, \delta(\ [c \mapsto T;\ \Sigma^k]\ , \&_J (c?(T_j); \varphi_b^j), \varphi_a^k) \,|\, \forall j \in J. \delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n)\}$$

then we can perform the reduction:

$$[c \mapsto T_k;\ \Sigma]\ \varphi_1 \parallel \ldots \parallel \&_J (c?(T_j);\ \varphi_b^j) \parallel \ldots \parallel \varphi_m \to [\Sigma]\ \varphi_1 \parallel \ldots \parallel \varphi_b^k \parallel \ldots \parallel \varphi_m$$

If it is of the form $\delta(\Sigma, \oplus_K (c!\langle T_k \rangle;\ \varphi_a^k), \&_J (c?(T_j);\ \varphi_b^j))$ then we can perform the reduction:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \oplus_N c!\langle T_n \rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m \to [\Sigma]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n \rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m$$

If it is of the form $\delta(\Sigma, (c!\langle T_k \rangle;\ \varphi_a^k), \&_J (c?(T_j);\ \varphi_b^j))$ then we can perform the reduction:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel c!\langle T_n \rangle;\ \varphi_a^n \parallel \ldots \parallel \varphi_m \to [\Sigma;\ c \mapsto T_n]\ \varphi_1 \parallel \ldots \parallel \varphi_a^n \parallel \ldots \parallel \varphi_m$$

Hence in each of the cases it is possible to reduce, as required.

$\square$

Using these lemmas we can prove that if a program point is in Δ then it is globally compatible:

Theorem 3.12. *Sufficiency of Global Session Types*

$$\forall \Sigma, \Phi.\ [\Sigma]\ \Phi \in \Delta \Rightarrow \exists k. \mathcal{G}^k(\Sigma, \Phi)$$

Proof. Suppose the hypothesis. We then prove the conclusion by induction over k in the definition of $\mathcal{G}^k(\Sigma, \Phi)$.

Consider the case where $k = 0$. By the definition of Δ we know that $\mathcal{G}^k(\Sigma', \Phi') \overset{\text{def}}{=} \texttt{true}$, and hence the implication $[\Sigma]\ \Phi \in \Delta \Rightarrow \mathcal{G}^0(\Sigma, \Phi)$ is trivially valid,

Consider the case where $k > 0$. By Figure 3.11 we know that

$$\mathcal{G}^k(\Sigma, \Phi) \overset{\text{def}}{=} \left[\begin{array}{l} [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \Rightarrow \mathcal{G}^{k-1}(\Sigma', \Phi') \\ \quad \wedge\ [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \vee \Phi = \Pi\, \epsilon \\ \quad \wedge \nexists \texttt{error} \in \Phi \end{array} \right.$$

In order to show that $\mathcal{G}^k(\Sigma, \Phi)$ holds we show that each of the constituents hold.

By Lemma 3.9 we know that, as $[\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi'$, then we have that $[\Sigma']\ \Phi' \in \Delta$. By the inductive step we know that $[\Sigma]\ \Phi \in \Delta \Rightarrow \mathcal{G}^l(\Sigma, \Phi)$, for all Σ and Φ and $l < k$, and hence that $\mathcal{G}^{k-1}(\Sigma', \Phi')$. We therefore have that the implication $[\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \Rightarrow \mathcal{G}^{k-1}(\Sigma', \Phi')$ holds.

By Lemma 3.11 we know that:

$$[\Sigma]\ \Phi \in \Delta \Rightarrow [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \vee \Phi \equiv \Pi\, \epsilon$$

By Lemma 3.10 we know that $\texttt{error} \notin \Phi$. By Lemma 3.9 we know that:

$$[\Sigma]\ \Phi \in \Delta \wedge [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \Rightarrow [\Sigma']\ \Phi' \in \Delta$$

which proves all the constituent conjuncts of $\mathcal{G}^k(\Sigma, \Phi)$, as required. $\square$

We reiterate the key safety and liveness theorems from (Bettini et al., 2008) as instantiations of Theorems 3.7 and 3.8, defining $\mathcal{C} \overset{\text{def}}{=} [\Sigma]\ \Phi$ using the fact that $[\Sigma]\ \Phi \in \Delta \Rightarrow \mathcal{G}(\Sigma, \Phi)$ (by Theorem 3.12).

Observation 3.13. Safety of Static Message Passing Programs

If $\vdash \boxed{[\sigma]}\ P$ using $\mathcal{C} \overset{\text{def}}{=} \boxed{[\Sigma]}\ \Phi \in \Delta$ and $\boxed{[\sigma]}\ P \to \boxed{[\sigma']}\ P'$ then $\nexists\texttt{error} \in P'$ and $\vdash \boxed{[\sigma']}\ P'$

Observation 3.14. Liveness of Static Message Passing Programs

If $\vdash \boxed{[\sigma]}\ P$ using $\mathcal{C} \overset{\text{def}}{=} \boxed{[\Sigma]}\ \Phi \in \Delta$ then $\boxed{[\sigma]}\ P \to \boxed{[\sigma']}\ P' \vee P = \Pi\, v$

3.5.2 Non-Blocking Message Passing

In order to increase asynchrony we may wish to permit receive actions to continue when they attempt to receive from an empty queue, rather than blocking and waiting for a value to become available. Whilst this asynchrony leads to less idling, the synchronisation behaviour required in order to avoid message passing errors is more complex. Consider the following program abstraction:

$$\boxed{[\Sigma_\emptyset]}\ c!\langle l_1\rangle;\ c!\langle T_1\rangle \oplus c!\langle l_2\rangle;\ c!\langle T_2\rangle \parallel c?(l_1);\ c?(T_1)\&c?(l_2);\ c?(T_2)\&c?(l_{[]});\ c?(T_3) \quad (3.41)$$

In this program the receiver presents two services to the sender. It also contains a third service, mapped to by $l_{[]}$; this is the value returned when a receive action is performed on an empty queue, to denote that case. We make use of a semantics that, upon performing a receive on an empty channel, puts a place-holder $[]$ into the queue that denotes that the next value sent on the queue should be immediately discarded. We suggest this as a possible semantics for non-blocking receives, which has not been considered in the session typing world, and as such is novel Consider a reduction sequence, where the receiver acts before the sender:

$$\begin{aligned}
&\boxed{[\Sigma_\emptyset]}\ c!\langle l_1\rangle;\ c!\langle T_1\rangle \oplus c!\langle l_2\rangle;\ c!\langle T_2\rangle \parallel c?(l_1);\ c?(T_1)\&c?(l_2);\ c?(T_2)\&c?(l_{[]});\ c?(T_3)\\
\to\ &\boxed{[c \mapsto []]}\ c!\langle l_1\rangle;\ c!\langle T_1\rangle \oplus c!\langle l_2\rangle;\ c!\langle T_2\rangle \parallel c?(T_3)\\
\to\ &\boxed{[c \mapsto []]}\ c!\langle l_1\rangle;\ c!\langle T_1\rangle \parallel c?(T_3)\\
\to\ &\boxed{[\Sigma_\emptyset]}\ c!\langle T_1\rangle \parallel c?(T_3)\\
\to\ &\boxed{[c \mapsto T_1]}\ \epsilon \parallel c?(T_3)\\
\to\ &\boxed{[\Sigma_\emptyset]}\ \epsilon \parallel \texttt{error}
\end{aligned} \quad (3.42)$$

Here, as the receiver chooses which service is to be performed, and the sender is not informed of this choice, it goes on to perform a different service, and a communication error can occur. In order to avoid such errors we must provide feedback to the sender, so that its control flow choices will mirror those of the receiver. In the non-blocking semantics, whenever a send action is performed we automatically return a unit value; this denotes that a send action always succeeds at adding a value on to the end of the relevant queue. We modify this semantics (Figure 3.15). The modified semantics makes use of queues that consist entirely of place-holders or entirely of typed messages (Figure 3.14). When a send is performed on a queue without any place-holders, a unit

$$\begin{array}{rcl} q_{[]} & ::= & [] \\ & | & \emptyset \\ & | & q_{[]}, q_{[]} \end{array}$$

Figure 3.14: Queues With Type Holes

$$\frac{}{\Sigma[c \mapsto q_{[]}] \xrightarrow{c?(T)} \Sigma[c \mapsto [], q_{[]}]} \qquad \frac{}{\Sigma[c \mapsto T', q] \xrightarrow{c?(T)} \Sigma[c \mapsto q]}$$

$$\frac{}{\Sigma[c \mapsto [], q_{[]}] \xrightarrow{c!\langle T\rangle} \Sigma[c \mapsto q_{[]}]} \qquad \frac{}{\Sigma[c \mapsto q] \xrightarrow{c!\langle T\rangle} \Sigma[c \mapsto q, T]}$$

$$\Sigma[c \mapsto q_{[]}](c?(T)) = l_{[]} \qquad \Sigma[c \mapsto T', q](c?(T)) = T$$

$$\Sigma[c \mapsto [], q_{[]}](c!\langle T\rangle) = l_{[]} \qquad \Sigma[c \mapsto q](c?(T)) = \textsc{Unit}$$

Figure 3.15: Semantics Of Non Blocking Message Queues

$$\begin{array}{lcl}
d_1 \to d_2 \colon r\langle \widetilde{T \mapsto G}\rangle_M \upharpoonright d_1 & \stackrel{\text{def}}{=} & \oplus_N((c!\langle T_n\rangle, \textsc{Unit}); G_n \upharpoonright d_1 \& (c!\langle T_n\rangle, l_{[]}); G_{l_{[]}} \upharpoonright d_1) \\
 & & \qquad N \neq \emptyset \wedge (N \cup l_{[]} \mapsto G_{l_{[]}}) \subseteq M \\
d_1 \to d_2 \colon r\langle \widetilde{T \mapsto G}\rangle \upharpoonright d_2 & \stackrel{\text{def}}{=} & \&_M(c?(T_m); G_m \upharpoonright d_1) \\
d_1 \to d_2 \colon r\langle \widetilde{T \mapsto G}\rangle \upharpoonright d & \stackrel{\text{def}}{=} & G_1 \upharpoonright d \\
(\mu \underline{X}.G') \upharpoonright d & \stackrel{\text{def}}{=} & \mu \underline{X}.(G' \upharpoonright d) \\
\underline{X} \upharpoonright d & \stackrel{\text{def}}{=} & \underline{X} \\
\mathsf{0} \upharpoonright d & \stackrel{\text{def}}{=} & \mathsf{0}
\end{array}$$

Figure 3.16: Global Session Types Projection For Blocking Message Passing

value is returned to indicate success, and that the receiver has not already passed this point. When a send is performed on a queue containing a place-holder [] the action returns $l_{[]}$ to indicate that this has occurred. Then, for each possible service choice, we create an *external* choice between an action that transmits the service choice where a unit value returned, followed by the sender's side of that service, and an action that transmits the service choice but returns a place-holder $l_{[]}$, followed by the service for $l_{[]}$. We demonstrate this with an example:

$$\begin{aligned}
& [\Sigma_\emptyset]\ ((\texttt{snd}(c,l_1),\textsc{Unit});\ c!\langle T_1\rangle)\&((\texttt{snd}(c,l_1),l_{[]});\ c!\langle T_3\rangle) \parallel c?(l_1);\ c?(T_1)\&c?(l_{[]});\ c?(T_3) \\
\to\ & [c \mapsto []]\ ((\texttt{snd}(c,l_1),\textsc{Unit});\ c!\langle T_1\rangle)\&((\texttt{snd}(c,l_1),l_{[]});\ c!\langle T_3\rangle) \parallel c?(T_3) \\
\to\ & [\Sigma_\emptyset]\ c!\langle T_3\rangle \parallel c?(T_3) \\
\to\ & [c \mapsto T_1]\ \epsilon \parallel c?(T_3) \\
\to\ & [\Sigma_\emptyset]\ \epsilon \parallel \epsilon
\end{aligned} \tag{3.43}$$

Note that, in order to avoid similar errors for the second send we would have to also replace the send $c!\langle T_3\rangle$ with $(\texttt{snd}(c,l_1),\textsc{Unit})\&(\texttt{snd}(c,l_1),l_{[]})$, the receive $c?(T_1)$ with $c?(T_1)\&c?(l_{[]})$, and the receive $c?(T_3)$ with $c?(T_3)\&c?(l_{[]})$; we omit these in order to simplify this example.

We define the updated projection function of Global Session Types for non-blocking message passing in Figure 3.16. We add an additional requirement to the well formedness of Global Session Types, namely that each send/receive pair must include the case of when $l_{[]}$ is received, as it can occur in any send/receive pair. Note that, in the projection $d_1 \to d_2\colon r\langle \widetilde{T \mapsto G}\rangle \upharpoonright d_2 \overset{\text{def}}{=} \&_M(c?(T_m);\ G_m \upharpoonright d_1)$, the $\widetilde{T \mapsto G}$ includes $\texttt{Lab}\, l_{[]} \mapsto G_{l_{[]}}$.

We construct the set of safe program points (Figure 3.17) in a similar way to how we constructed the set for blocking message passing. The base case consists of effects that are projections of Global Session Types, alongside empty channel queues. The first half of the inductive case is similar to that in Figure 3.13, adding a message to the front of the queue and performing an external choice at the receiver. Note that this construction is only permitted if there are no place-holders in the queue that is being extended. The second half of the inductive case is formalises the approach described above of providing external choice at the receiver. A place-holder can be placed at the front of some message queues assuming no actual messages are in the queue being extended (but place-holders may be in the queue). The extended effect provides an external choice between a send action that receives a place-holder, indicating that the receiver has already skipped over this send, and and a send action that receives the unit value, indicating that the message has been successfully added to the relevant queue.

We can prove to show that if a program point is in the valid points set, and if the point reduces, then it stays in the valid points set.

Lemma 3.15. *Valid Points Set Reduction*

$$\forall \Sigma, \Phi, n, \Sigma', \Phi'.\ [\Sigma]\ \Phi \in \Delta_n \wedge [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow [\Sigma']\ \Phi' \in \Delta_{n+1}$$

Proof. The proof proceeds similarly to that in Lemma 3.9. Suppose the hypotheses. We then prove the conclusion by induction over the n of Δ_n. The key differences are in the case where $n > 0$. By the structure of parallel effects we can identify one effect which is

$$\Delta \overset{\text{def}}{=} \bigcup_{n<\omega} \Delta_n \qquad \Delta_0 \overset{\text{def}}{=} \{\, [\Sigma_\emptyset]\ \Phi^{\restriction G}\}$$

$$\begin{aligned}
\Delta_{n+1} \overset{\text{def}}{=} \quad & \{\, \delta(\ [c \mapsto T;\ \Sigma^k]\ , \&_J(c?(T_j);\ \varphi_b^j), \varphi_a^k) \mid \forall j \in J . \delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n)\} \\
& \cup \{\, \delta(\ [\Sigma]\ , \oplus_K(c!\langle T_k\rangle;\ \varphi_a^k), \&_J(c?(T_j);\ \varphi_b^j) \mid \\
& \qquad \forall k \in K \,.\, \delta(\Sigma, \varphi_a^k, \varphi_b^k) \in \Delta_n \ \wedge\ \texttt{Lab}\ l_{[]} \in \widetilde{T}_K \ \wedge\ \Sigma(c) = \emptyset) \\
& \cup \{\, \delta(\ [c \mapsto [];\ \Sigma^k]\ , (\oplus_J((c!\langle T_j\rangle, \textsc{Unit});\ \phi_a^j))\&((c?(T_k), l_{[]});\ \phi_a^k)) \mid \\
& \qquad \forall j \in J . \delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n)\} \\
& \cup \Delta_n
\end{aligned}$$

where $c = d_a d_b$ and $\emptyset \subset K \subseteq J$.

Figure 3.17: Non Blocking Message Passing Safe Program Points Set

the one that reduces:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi_m \rightarrow [\Sigma']\ \varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \varphi_m$$

As $[\Sigma]\ \Phi \in \Delta_n$ we know that either in one of the constructions in the definition of Δ_n, or it is in Δ_{n-1}. The case when $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \&_J(c?(T_j);\ \varphi_b^j))\}$ is the same as in Lemma 3.9.

Consider the case when $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \oplus_K(c!\langle T_k\rangle;\ \varphi_a^k), \&_J(c?(T_j);\ \varphi_b^j))\}$ where $\forall j \in J.\delta(\Sigma^j, \varphi_a^j, \varphi_b^j) \in \Delta_{n-1}$ and $l_{[]} \in \widetilde{T}_J$ and $\Sigma(c) = \emptyset$ (as in Figure 3.17). We can perform a case split over the index i. If $i = b$ then, by the reduction rules we know that $\Sigma' = \Sigma; c \mapsto l_{[]}$ and $\varphi_b' = \varphi_b^{l_{[]}}$. By the third part of the definition of Δ_{n+1} we can show that $\delta([c \mapsto []; \Sigma^k], \oplus_J((c!\langle T_j\rangle, \textsc{Unit});\ \varphi_a^j)\&((c?(T_{l_{[]}}), l_{[]});\ \varphi_a^{l_{[]}}), \varphi_b^{l_{[]}}) \in \Delta_n$, as by the hypothesis we know that $\forall j \in J.\delta(\Sigma^j, \varphi_a^j, \varphi_b^j) \in \Delta_n$. Consider the case when $i = a$. If $\oplus_K(c!\langle T_k\rangle;\ \varphi_a^k)$ is a choice then we can use the internal choice reduction rules to show that:

$$[\Sigma]\ \varphi_1 \parallel \ldots \parallel \oplus_K(c!\langle T_k\rangle;\ \varphi_a^k) \parallel \ldots \parallel \varphi_m \rightarrow [\Sigma]\ \varphi_1 \parallel \ldots \parallel \oplus_L(c!\langle T_l\rangle;\ \varphi_a^l) \parallel \ldots \parallel \varphi_m$$

for some L such that $\emptyset \subset L \subseteq J$. We straightforwardly have that $\varphi_1 \parallel \ldots \parallel \oplus_L(c!\langle T_l\rangle;\ \varphi_a^l) \parallel \ldots \parallel \varphi_m \in \Delta_n$. If $\oplus_K(c!\langle T_k\rangle;\ \varphi_a^k)$ is a vacuous choice then it is simply of the form $c!\langle T_k\rangle;\ \varphi_a^k$. By the reduction rules then we know that $\Sigma' = \Sigma_k;\ c \mapsto T_k$ and $\varphi_a' = \varphi_a^k$. By the hypotheses we know that $\delta(\Sigma^k, \varphi_a^k, \varphi_b^k) \in \Delta_{n-1}$. Then, by the first part of the definition of $\Delta_n^{\restriction G}$ in Figure 3.17 we know that $[\Sigma']\ \Phi' = \delta([c \mapsto T; \Sigma^k], \varphi_a^k, \&_J(c?(T_j);\ \varphi_b^j)) \in \Delta_n^{\restriction G}$ as required. Consider the case when $i \neq b$ and $i \neq a$. This follows similarly to the case where $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \varphi_a^k, \&_J(c?(T_j);\ \varphi_b^j))\}$ and $i \neq b$, above.

Consider the case where, given $\forall j \in J.\delta(\Sigma^j, \varphi_a^j, \varphi_b^j) \in \Delta_{n-1}$, that $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto []; \Sigma^k], \oplus_J((c!\langle T_j\rangle, \textsc{Unit});\ \varphi_a^j)\&((c?(T_k), l_{[]});\ \varphi_a^k))\ \}$. We can perform a case split over the index i. Consider the case where $i = a$. By the reduction rules we know that $\Sigma' = \Sigma_k$

and $\varphi'_a = \varphi^k_a$. By the hypotheses we know that $\delta(\Sigma^k, \varphi^k_a, \varphi^k_b) \in \Delta_{n-1}$, and hence that $\delta(\Sigma^k, \varphi^k_a, \varphi^k_b) \in \Delta_n$ as required. Consider the case when $i \neq a$. This case is similar to the case where $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \&_J(c?(T_j);\ \varphi^j_b))\}$ and $i \neq b$, above.

Consider the case when $[\Sigma]\ \Phi \in \Delta_{n-1}$. By the inductive hypothesis we have that $[\Sigma']\ \Phi' \in \Delta_{n-1}$, and hence that $[\Sigma']\ \Phi' \in \Delta_n$ as required. $\square$

We can also prove relatively straightforwardly that if program point is in Δ then it contains no errors, and that it is live.

Lemma 3.16. *Non Blocking Valid States Safety*

$$\forall \Sigma, \Phi, n.\ [\Sigma]\ \Phi \in \Delta_n \Rightarrow \mathtt{error} \notin \Phi$$

Proof. Suppose the hypothesis. We then prove the conclusion by induction over n. $\square$

Lemma 3.17. *Non Blocking Valid States Liveness*

$$\forall \Sigma, \Phi, n.\ [\Sigma]\ \Phi \in \Delta_n \Rightarrow \exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \rightarrow [\Sigma']\ \Phi' \vee \Phi \equiv \Pi\, \epsilon$$

Proof. Suppose the hypothesis. We then prove that one or other of the conclusions are true, by induction over the n of Δ_n. This proof essentially follows that in Lemma 3.11. The key difference is when $n > 0$ and

$$[\Sigma]\ \Phi \in \{\ \delta([c \mapsto []; \Sigma^k], \oplus_J((c!\langle T_j \rangle, \textsc{Unit});\ \varphi^j_a)\&((c?(T_k), l_{[]});\ \varphi^k_a))\}$$

where $\forall j \in J. \delta(\Sigma^j, \varphi^j_a) \in \Delta_n$. By the reduction rules we know that:

$$[c \mapsto []; \Sigma^k]\, c?(T_k) \rightarrow [\Sigma^k]\, \epsilon$$

and hence that

$$\delta([c \mapsto []; \Sigma^k], \oplus_J((c!\langle T_j \rangle, \textsc{Unit});\ \varphi^j_a)\&((c?(T_k), l_{[]});\ \varphi^k_a)) \rightarrow \delta(\Sigma^k, \phi^k_a)$$

as required. $\square$

Using these lemmas we can prove that if a program point is in Δ then it is globally compatible:

Theorem 3.18. *Sufficiency of Global Session Types*

$$\forall \Sigma, \Phi.\ [\Sigma]\ \Phi \in \Delta \Rightarrow \exists k. \mathcal{G}^k(\Sigma, \Phi)$$

Proof. This proof is similar to that of Theorem 3.12, except for it uses Lemmas 3.15, 3.16 and 3.17 rather than Lemmas 3.9, 3.10 and 3.11. $\square$

We can hence guarantee an absence of communication errors in a message passing system with very little synchronisation. The synchronisation behaviour for blocking message passing relies on the fact that the participant that provides a service must wait for the participant that invokes the service to choose which service will be performed. When the participant that provides a service can choose which service will proceed, the participant that invokes the service must similarly be informed of the decision, to ensure that both participants proceed down the same service path. The proofs of safety and liveness for blocking message passing make use of an intuitive, inductive definition that determines which states are acceptable (those that can be reached by reduction from the projection of a Global Session Type). When we use this definition, the portions of the proofs that use the synchronisation behaviour are immediately apparent. Hence we can reuse the structure and large portions of the proof for blocking message passing.

3.6 Conclusions

In this chapter we consider communication safety and liveness for functional programs which access accompanying shared state with an arbitrary semantics. Our contributions are as follows. We extend previous work where only receive actions could return values of different types to a system where all accesses to the shared state can return typed values. We show how to prove safety and liveness in a system where the shared state has arbitrary semantics. We reprove existing results of safety and liveness for blocking message passing programs, using a novel, inductive definition, which we believe is significantly more comprehensible. We extend this technique to cover non-blocking message passing systems, and show how our inductive definition can be straightforwardly modified to cover non-blocking message passing.

Chapter 4

Dynamic Software Update

In this chapter we consider how to update message passing programs, whilst maintaining safety and liveness properties. We explore communications safety and deadlock freedom for updatable asynchronous message passing programs with blocking receives, and provide examples of erroneous updates to programs, and reductions of such systems (Section 4.1). We provide a calculus for updates to the language presented in Chapter 3, an operational semantics for update, and extend our behavioural abstraction (Section 4.2). We provide a general, inductive description of the communication safety and deadlock freedom properties in the presence of arbitrary updates (Section 4.3). We present our conclusions in Section 4.4

$$\begin{array}{l} t_c = \mu\underline{X}.\texttt{sel}(c_1, \texttt{prime?}).\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Int}).\underline{X} \\ t_s = \mu\underline{X}'.\texttt{case}\ c_1\ \texttt{of}\ \{\ \begin{array}[t]{lll} \texttt{prime?} & \mapsto & \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \\ \ldots\} \end{array} \end{array}$$

Figure 4.1: Maths Server

$$\begin{array}{l} t_c' = \mu\underline{X}.\texttt{sel}(c_1, \texttt{prime?}).\texttt{snd}(c_1, \ldots).\texttt{case}\ c_2\ \texttt{of}\ \{\ \begin{array}[t]{lll} \texttt{result} & \mapsto & \texttt{rcv}(c_2)(x : \textsc{Int}).\underline{X}, \\ \texttt{err} & \mapsto & \underline{X}\} \end{array} \\ t_s' = \mu\underline{X}'.\texttt{case}\ c_1\ \texttt{of}\ \{\ \begin{array}[t]{lll} \texttt{prime?} & \mapsto & \texttt{rcv}(c_1)(x : \textsc{Int}). \\ & & \texttt{if}\ x < 3000\ \texttt{then} \\ & & \quad \texttt{sel}(c_2, \texttt{result}).\texttt{snd}(c_2, \ldots).\underline{X}' \\ & & \texttt{else} \\ & & \quad \texttt{sel}(c_2, \texttt{err}).\underline{X}' \\ \ldots\} \end{array} \end{array}$$

Figure 4.2: Updated Maths Server

1 $[\sigma_\emptyset]\ (t_c \parallel t_s)$

2 $\quad \rightarrow \ldots$

3 $\quad \rightarrow [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t_c \parallel t_s)$

4 $\quad \xrightarrow{\psi} [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t'_c \parallel t'_s)$

5 $\quad \equiv [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t'_c \parallel$
$\qquad \texttt{if}\ x < 3000\ \texttt{then}\ \texttt{sel}(c_2, \texttt{result}).\texttt{snd}(c_2, \ldots).t'_s\ \texttt{else}\ \texttt{sel}(c_2, \texttt{err}).t'_s)$

6 $\quad \rightarrow [c_1 \mapsto 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t'_c \parallel \texttt{rcv}(c_1)(x : \textsc{Int}).$
$\qquad \texttt{if}\ x < 3000\ \texttt{then}\ \texttt{sel}(c_2, \texttt{result}).\texttt{snd}(c_2, \ldots).t'_s\ \texttt{else}\ \texttt{sel}(c_2, \texttt{err}).t'_s)$

7 $\quad \rightarrow \ldots$

8 $\quad \rightarrow [\sigma_\emptyset]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t'_c \parallel \texttt{sel}(c_2, \texttt{result}).\texttt{snd}(c_2, \ldots).t'_s)$

9 $\quad \rightarrow [c_2 \mapsto \texttt{result}]\ (\texttt{rcv}(c_2)(x : \textsc{Bool}).t'_c \parallel \texttt{snd}(c, \ldots).t'_s)$

10 $\quad \rightarrow [\sigma_\emptyset]\ (\texttt{error} \parallel \texttt{snd}(c_2, \ldots).t'_s)$

Figure 4.3: Update Error Due To Update At Mismatched Code Points

4.1 Motivation: Message Passing Programs

We specify a maths server program in Figure 4.1. In this program the client t_c tells the server t_s that it would like the server to tell it if a given number is a prime. The client then sends the number to the server, and the server will send an integer reply. When the program is deployed it may be discovered that the server is often encountering errors due to overflow or timeouts. We may then wish to modify the program so that the server can choose to abort the calculation, and inform the client whether the computation produced a result or an error. We specify such a program in Figure 4.2. In order to migrate from one program to another we need some mechanism to modify the running code. We delay the full specification of this mechanism until Section 4.2 and, for the purposes of the motivation, describe updates informally.

The introduction of an update at an inopportune point can introduce a communication error into an otherwise safe program. Consider the reduction shown in Figure 4.3. The original maths server reduces until line 3. Then an update reduction, which is annotated with ψ, occurs on line 4; this modifies the continuations of both the client and the server to be the bodies of the updated maths server. Note that at line 3 the client has entered the body of its recursive loop, whilst the server has not. As the update occurs at this point we end up with a partial reduction of the old client code attempting to interact with the new version of the server; hence the server sends the `result` label to indicate that it has completed the computation successfully. Unfortunately the client is only expecting a boolean result, rather than a label and a result, and hence a communication error occurs. This illustrates how messages in the queue at the point an update is applied can affect whether the updated program is safe.

We specify a Producer/Consumer program, with an acknowledgement, in Figure 4.4. In this program the producer t_p sends one data item to the consumer t_{co} , and waits for an acknowledgement from the consumer that indicates that it is ready to continue. We may then wish to modify the program to permit the producer to send two data items before waiting for the acknowledgement, in order to increase asynchrony. We specify such a program in Figure 4.5.

The introduction of an update at an inopportune point can also introduce deadlock into an otherwise live program. Consider a reduction of the Producer/Consumer example, shown in Figure 4.6. The original Producer/Consumer program runs until line `3`. Then an update reduction occurs on line `4`; this modifies the continuations of both the client and the server to be the bodies of the updated Producer/Consumer program. Note that at line `3` the producer has entered the body of its recursive loop, while the consumer has not. As the update occurs at this point we end up with a partial reduction of the old producer attempting to interact with the new version of the server. The producer hence sends one data value and waits for an acknowledgement whilst the consumer receives the first data value and waits to receive the second before it sends the acknowledgement. This leads to a deadlocked system, where the producer and the consumer are both waiting for a message from the other.

$$
\begin{aligned}
t_p &= \mu\underline{X}.\texttt{snd}(c_1, v_1).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X} \\
t_{co} &= \mu\underline{X}'.\texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).\underline{X}'
\end{aligned}
$$

Figure 4.4: Producer/Consumer System With Acknowledgement

$$
\begin{aligned}
t'_p &= \mu\underline{X}.\texttt{snd}(c_1, v_1).\texttt{snd}(c_1, v_2).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X} \\
t'_{co} &= \mu\underline{X}'.\texttt{rcv}(c_1)(x : T_1).\texttt{rcv}(c_1)(x : T_2).\texttt{snd}(c_2, ()).\underline{X}'
\end{aligned}
$$

Figure 4.5: Updated Producer/Consumer System With Acknowledgement

$$
\begin{array}{ll}
1 & [\sigma_\emptyset]\ (t_p \parallel t_{co}) \\
2 & \quad \rightarrow \ldots \\
3 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{snd}(c_1, v_1).\texttt{rcv}(c_2)(x : \textsc{Unit}).t_p \parallel t_{co}) \\
4 & \quad \xrightarrow{\psi} [\sigma_\emptyset]\ (\texttt{snd}(c_1, v_1).\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel t'_{co}) \\
5 & \quad \rightarrow [c_1 \mapsto v_1]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel t'_{co}) \\
6 & \quad \rightarrow [c_1 \mapsto v_1]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{rcv}(c_1)(x : T_2).\texttt{snd}(c_2, ()).t'_{co}) \\
7 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{rcv}(c_1)(x : T_2).\texttt{snd}(c_2, ()).t'_{co}) \\
8 & \quad \not\rightarrow
\end{array}
$$

Figure 4.6: Update Deadlock Due To Update At Mismatched Code Points

$$
\begin{array}{ll}
1 & [\sigma_\emptyset]\ (t_p \parallel t_{co}) \\
2 & \quad \rightarrow \ldots \\
3 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{snd}(c_1, v_1).\texttt{rcv}(c_2)(x : \textsc{Unit}).t_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).t_{co}) \\
4 & \quad \xrightarrow{\psi} [\sigma_\emptyset]\ (\texttt{snd}(c_1, v_1).\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).t_{co}) \\
5 & \quad \rightarrow \ldots \\
6 & \quad \rightarrow [\sigma_\emptyset]\ (t'_p \parallel t_{co}) \\
7 & \quad \rightarrow \ldots \\
8 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{snd}(c_1, v_2).\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{snd}(c_2, ()).t_{co}) \\
9 & \quad \rightarrow \ldots \\
10 & \quad \rightarrow [c_1 \mapsto v_2, c_2 \mapsto ()]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).t_{co}) \\
11 & \quad \rightarrow [c_2 \mapsto ()]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t'_p \parallel \texttt{error})
\end{array}
$$

Figure 4.7: Update Error Due To Inconsistent Update

We also wish to rule out trivial update errors. Consider an update that modifies the code of the producer to t'_p but leaves the code of the consumer as t_{co}. We present one reduction of this system in Figure 4.7. We apply the update on line 4. The system then reduces to complete one loop of each thread's main loop. When the producer starts a new loop it uses the new code (line 6). The sender sends both its values, and the consumer performs two loops. On its second loop the consumer receives v_2, which is not of the type it is expecting.

It is, however, possible to safely perform the updates to the maths server and the Producer/Consumer system. Consider again the maths server and its update (Figures 4.1 & 4.2). We present a possible reduction which leaves the program both safe and live after the update occurs (Figure 4.8). The original maths server starts reducing; on line 6, part way through each thread's recursive loop, the update is applied. Since the server and the client are each part way through their old code's loop body they both continue communicating using the old code until they reach the point the point where they would recurse (line 11), at which point each continues using the new code.

To summarise, the effect of the code, and the messages in queue, at the point an update is applied will determine the safety and liveness of the updated program. In particular, all threads must update when they are on the same 'run' of an protocol. Possible techniques to achieve this include ensuring that all participants are always on the same run of a protocol, or to determine the run of the participant that is the furthest ahead, and to update each participant when it reaches that point.

$$\begin{array}{lll}
1 & & [\sigma_\emptyset]\ (t_c \parallel t_s) \\
2 & \rightarrow & \ldots \\
3 & \rightarrow & [c_1 \mapsto \texttt{prime?}]\ (\texttt{snd}(c, 3457).\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
 & & \quad \mu\underline{X}'.\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \ldots\}) \\
4 & \rightarrow & [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
 & & \quad \mu\underline{X}'.\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \ldots\}) \\
5 & \rightarrow & [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
 & & \quad \texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t_s, \ldots\}) \\
6 & \xrightarrow{\psi} & [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel \\
 & & \quad \texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t'_s, \ldots\}) \\
7 & \rightarrow & [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel \\
 & & \quad \texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t'_s, \ldots\}) \\
8 & \rightarrow & [c_1 \mapsto 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t'_s) \\
9 & \rightarrow & [\sigma_\emptyset]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel \texttt{snd}(c_2, \ldots).t'_s) \\
10 & \rightarrow & [c_2 \mapsto \texttt{true}]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel t'_s) \\
11 & \rightarrow & [\sigma_\emptyset]\ (t'_c \parallel t'_s) \\
12 & \rightarrow & \ldots
\end{array}$$

Figure 4.8: Update Success In Maths Server

4.2 Update Calculus

In order to reason about how and when updates occur we introduce two additions to our calculus (Figure 4.9). We augment our language from Figure 3.4 with terms used to describe where updates can occur. We define our notion of updates, which describe the changes to a running program, and the program points at which said changes can be made. We also extend our behavioural abstraction to encompass updates.

4.2.1 Language Additions

A region annotation $f(t)$ denotes a region or body of code, annotated with the name f, which can be replaced in its entirety. Such regions normally consist of a procedure or a function, but could be used to consider conceptual units of work, for example a series of processing calls, or a series of logging calls. When reducing this region, in the absence of updates, the annotation is stripped away as the region proper is reduced. Once the annotation has been stripped away the body cannot be replaced (though it may be possible to replace further annotated regions within the body). Updates replace the body of code inside an annotation with another body of code. This corresponds with our intuition that the body of the annotated region can only be replaced in its entirety. Annotations are a variant of the standard Dynamic Software Update approach which

$$
\begin{array}{lcll}
t & ::= & & \textit{Terms} \\
 & | & \ldots & \\
 & | & f(t) & \text{updatable regions} \\
 & | & \texttt{dmod}_{\widetilde{f}}(t) & \text{transactional regions} \\
 & & & \\
\psi,\omega & ::= & & \textit{Predicated Updates} \\
 & | & p \mapsto u & \text{Map from predicate to update} \\
 & | & \psi,\psi & \text{List of predicated updates} \\
 & | & \emptyset & \text{Empty predicated update} \\
 & & & \\
u & ::= & & \textit{Region updates} \\
 & | & f \mapsto t & \text{Map from region} \\
 & & & \text{to new code} \\
 & | & u,u & \text{Set of updates} \\
 & | & \emptyset & \text{Empty update} \\
 & & & \\
p & ::= & & \textit{Update predicates} \\
 & | & p(\sigma,P) & \textit{on code and message queues}
\end{array}
$$

Figure 4.9: Update Calculus

$$
\frac{[\sigma]\ t \xrightarrow{\tau}_F [\sigma']\ t'}{[\sigma']\ f(t) \xrightarrow{\tau}_F [\sigma']\ f(t')}
\qquad
\frac{[\sigma]\ t \xrightarrow{\alpha(\widetilde{v})}_F [\sigma']\ t'}{[\sigma]\ f(t) \xrightarrow{\alpha(\widetilde{v})}_{F\cup\{f\}} [\sigma']\ t'}
\qquad
\frac{}{[\sigma]\ f(v) \xrightarrow{\tau} [\sigma]\ v}
$$

$$
\frac{[\sigma]\ t \xrightarrow{\gamma}_{F'} [\sigma']\ t' \qquad \texttt{regions}(t) \not\subseteq F}{[\sigma]\ \texttt{dmod}_F(t) \xrightarrow{\gamma}_{F'} [\sigma']\ \texttt{dmod}_{F\cup F'}(t')}
\qquad
\frac{\texttt{regions}(t) \subseteq F}{[\sigma]\ \texttt{dmod}_F(t) \xrightarrow{\tau} [\sigma]\ t}
$$

$$
\frac{[\sigma]\ P \xrightarrow{\gamma}_F [\sigma']\ P' \qquad \nexists p \in \mathsf{dom}(\psi).p(\sigma,P)}{[\sigma]\ P,\psi \xrightarrow{\gamma} [\sigma']\ P',\psi}
\qquad
\frac{}{[\sigma]\ P,\psi \xrightarrow{\tau} [\sigma]\ P,(\psi,\omega)}
$$

$$
\frac{\psi = p_1 \mapsto u_1,\ldots,p_n \mapsto u_n \qquad i \text{ smallest in } 1...\text{n}.p_i(\sigma,P)}{[\sigma]\ P,\psi \xrightarrow{\tau} [\sigma]\ \mathsf{upd}(P,u_i,\texttt{false}),\psi \setminus p_i \mapsto u_i}
$$

Figure 4.10: Update Calculus Operational Semantics

permits the bodies of functions to be changed in their entirety (Hicks, 2005; Stoyle et al., 2005). The standard approach holds the body of a function in the heap and substitutes it for the function whenever a call is made. Instead, we simply annotate the code itself.

A transactional annotation $\texttt{dmod}_{\widetilde{f}}(t)$ denotes a region of code where, if an update modifies more than one annotated region, that it must be able to modify all annotated regions within the transactional annotation, or none of them. The subscript $\widetilde{f}$ records the annotations of named regions which have already been entered, by reducing the body

$$
\begin{array}{lcll}
\mathsf{upd}(v, u, b) & \overset{\text{def}}{=} & v & v \neq \mathtt{rec}\,\underline{X}(x:T).t \\
\mathsf{upd}((\alpha(\widetilde{v}), T), u, b) & \overset{\text{def}}{=} & (\alpha(\widetilde{v}), T) & \\
\mathsf{upd}(\mathtt{rec}\,\underline{X}(x:T).t, u, b) & \overset{\text{def}}{=} & \mathtt{rec}\,\underline{X}(x:T).\mathsf{upd}(t, u, b) & \\
\mathsf{upd}(t_1\, t_2, u, b) & \overset{\text{def}}{=} & \mathsf{upd}(t_1, u, b)\; \mathsf{upd}(t_2, u, b) & \\
\mathsf{upd}(\mathtt{if}\, t_1\, \mathtt{then}\, t_2\, \mathtt{else}\, t_3, u, b) & \overset{\text{def}}{=} & \mathtt{if}\, t_1'\, \mathtt{then}\, t_2'\, \mathtt{else}\, t_3' & t_i' = \mathsf{upd}(t_i, u, b) \\
\mathsf{upd}(\mathtt{case}\, \alpha(\widetilde{v})\, \mathtt{in}\, \{\widetilde{T \mapsto t}\}, u, b) & \overset{\text{def}}{=} & \mathtt{case}\, \alpha(\widetilde{v})\, \mathtt{in}\, \{\widetilde{T \mapsto \mathsf{upd}(t_1, u, b)}\} & \\
\mathsf{upd}(\mathtt{dmod}_F(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_F(\mathsf{upd}(t_1, u, \mathtt{true})) & F \cap \mathsf{dom}(u) = \emptyset \\
\mathsf{upd}(\mathtt{dmod}_F(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_F(\mathsf{upd}(t_1, u, \mathtt{false})) & F \cap \mathsf{dom}(u) \neq \emptyset \\
\mathsf{upd}(f(t_1), u, b) & \overset{\text{def}}{=} & f(\mathsf{upd}(t_1, u, b)) & \neg b \vee \nexists t. \\
 & & & \quad (f \mapsto t) \in u \\
\mathsf{upd}(f(t_1), u, \mathtt{true}) & \overset{\text{def}}{=} & f(t') & (f \mapsto t') \in u
\end{array}
$$

Figure 4.11: Update Function

$$
\begin{array}{ll}
1 & [\_]\ \mathtt{dmod}_{\emptyset}(\ldots;\ f(t_1);\ \ldots;\ g(t_2)) \\
2 & \quad \rightarrow \ldots \\
3 & \quad \rightarrow [\_]\ \mathtt{dmod}_{\emptyset}(f(t_1);\ \ldots;\ g(t_2)) \\
4 & \quad \rightarrow [\_]\ \mathtt{dmod}_{f}(t_1;\ \ldots;\ g(t_2)) \\
5 & \quad \rightarrow \ldots \\
6 & \quad \rightarrow [\_]\ \mathtt{dmod}_{f}(g(t_2)) \\
7 & \quad \rightarrow [\_]\ \mathtt{dmod}_{f,g}(t_2) \\
9 & \quad \rightarrow [\_]\ t_2 \\
8 & \quad \rightarrow \ldots
\end{array}
$$

Figure 4.12: updatable region Reduction Example

of the *transactional region*. We explain using the reduction in Figure 4.12. When an *updatable region* (named region) is reduced inside a transactional region its region name is added to the set of updatable regions which cannot be updated within the transactional region (lines 4 and 7). An update to f can occur at any point up to and including line 3, an update to g can occur at any point up to and including line 6, and an update to both f and g can occur at any point up to and including line 3. If a transactional region is nested within another transactional region it is handled separately to the body of the encapsulating region. In order to inform encapsulating transactional regions that an updatable region has been entered we annotate our reduction rules; compound operators pass through reductions of sub-terms, and all destructive operator reductions are annotated with τ except the reduction for updatable regions.

In order to be able to update the bodies of the recursive loops of the Producer/Consumer example, as we did on line 4 of Figure 4.6, in our full calculus we write the original

$$\begin{array}{lll}
1 & [\sigma_\emptyset]\ (t_p \parallel t_{co}), \psi_\emptyset & \\
2 & \quad \rightarrow \ \ldots & \\
3 & \quad \rightarrow \ [c_1 \mapsto v_1]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).t_{co}), \psi_\emptyset & \\
4 & \quad \rightarrow \ [c_1 \mapsto v_1]\ (\texttt{rcv}(c_2)(x : \textsc{Unit}).t_p \parallel \texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).t_{co}), \psi & \\
5 & \quad \rightarrow \ \ldots & \\
6 & \quad \rightarrow \ [\sigma_\emptyset]\ (\mu \underline{X}.\texttt{dmod}_\emptyset(f(\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X})) \parallel & \\
 & \qquad \mu \underline{X}'.\texttt{dmod}_\emptyset(g(\texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).\underline{X}'))), \psi & \\
7 & \quad \rightarrow \ [\sigma_\emptyset]\ (\mu \underline{X}.\texttt{dmod}_\emptyset(f(\mu \underline{X}''.\texttt{snd}(c_1, \ldots).\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X}'')) \parallel & \\
 & \qquad \mu \underline{X}'.\texttt{dmod}_\emptyset(g(\mu \underline{X}'''.\texttt{rcv}(c_1)(x : T_1).\texttt{rcv}(c_1)(x : T_2).\texttt{snd}(c_2, ()).\underline{X}'''))), \psi_\emptyset & \\
8 & \quad \rightarrow \ \ldots &
\end{array}$$

$$\psi \overset{\text{def}}{=} p \mapsto u \qquad p(\sigma, P) \overset{\text{def}}{=} (P = t_p \parallel t_{co}) \qquad u \overset{\text{def}}{=} f \mapsto t'_p, f \mapsto t'_{co}$$

$$\begin{aligned}
t_p &= \mu \underline{X}.\texttt{dmod}_\emptyset(f(\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X})) \\
t_{co} &= \mu \underline{X}'.\texttt{dmod}_\emptyset(g(\texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).\underline{X}'))
\end{aligned}$$

$$\begin{aligned}
t'_p &= \mu \underline{X}''.\texttt{snd}(c_1, \ldots).\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X}'' \\
t'_{co} &= \mu \underline{X}'''.\texttt{rcv}(c_1)(x : T_1).\texttt{rcv}(c_1)(x : T_2).\texttt{snd}(c_2, ()).\underline{X}'''
\end{aligned}$$

Figure 4.13: Example Update Reduction

---

program as:

$$\begin{aligned}
t_p &= \mu \underline{X}.\texttt{dmod}_\emptyset(f(\texttt{snd}(c_1, \ldots).\texttt{rcv}(c_2)(x : \textsc{Unit}).\underline{X})) \\
t_{co} &= \mu \underline{X}'.\texttt{dmod}_\emptyset(g(\texttt{rcv}(c_1)(x : T_1).\texttt{snd}(c_2, ()).\underline{X}'))
\end{aligned} \tag{4.1}$$

4.2.2 Updates

In Figure 4.9 we present our definition of updates. *Region updates* u are maps from region names f to new region bodies t. *Predicated updates* ψ are a list of mappings from *update predicates* to region updates. An update predicate $p(\sigma, P)$ is a first order logic predicate, over the structure of the code, the shared state, and the structure of the predicated updates themselves. Update predicates denote when it is appropriate to apply a given region update to the existing program. Example predicates include:

$$p(\sigma, P) \overset{\text{def}}{=} (\sigma = \sigma_\emptyset) \qquad p(\sigma, P) \overset{\text{def}}{=} (P = t_p \parallel t_{co}) \tag{4.2}$$

A region update is applied to the code as soon as its associated update predicate becomes true:

$$\frac{\psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n \qquad i \text{ smallest in } 1\ldots\text{n}.p_i(\sigma, P)}{[\sigma]\ P, \psi \xrightarrow{\tau} [\sigma]\ \texttt{upd}(P, u_i, \texttt{false}), \psi \setminus p_i \mapsto u_i} \tag{4.3}$$

The order of the maps from predicates to region updates is important, as updates are applied from left to right (oldest to newest). We define an update function $\mathsf{upd}(t, u, b)$ (Figure 4.11) where b denotes whether t occurs in a context which is appropriate to update. The definition of update for transactional regions describes in which situations it is appropriate to update:

$$\begin{array}{lcll} \mathsf{upd}(\mathtt{dmod}_{\widetilde{f}}(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_{\widetilde{f}}(\mathsf{upd}(t_1, u, \mathtt{true})) & \widetilde{f} \cap \mathsf{dom}(u) = \emptyset \\ \mathsf{upd}(\mathtt{dmod}_{\widetilde{f}}(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_{\widetilde{f}}(\mathsf{upd}(t_1, u, \mathtt{false})) & \widetilde{f} \cap \mathsf{dom}(u) \neq \emptyset \end{array} \tag{4.4}$$

When no updatable regions (that are being updated by u) have been entered, within a given transactional region, then it is appropriate to update all possible named regions within the transactional region, otherwise it is not. When an update is applied to a named region, if we are in an appropriate context, and the update contains a replacement body for the named region, then we replace the body of the region in its entirety:

$$\mathsf{upd}(f(t_1), u, \mathtt{true}) \quad \overset{\text{def}}{=} \quad f(u(f)) \quad f \in \mathsf{dom}(u) \tag{4.5}$$

Otherwise we continue recursively:

$$\mathsf{upd}(f(t_1), u, b) \quad \overset{\text{def}}{=} \quad f(\mathsf{upd}(t_1, u, b)) \quad (\neg b \vee f \notin \mathsf{dom}(u)) \tag{4.6}$$

Updates can be introduced at any time, and are always appended onto the end of the update list:

$$\frac{}{\boxed{[\sigma]}\ P, \psi \xrightarrow{\tau} \boxed{[\sigma]}\ P, (\psi, \omega)} \tag{4.7}$$

Here ω is introduced externally. An external, or out-of-band, update (Soules et al., 2003; Ajmani, 2004; Stoyle et al., 2005; Nicoara et al., 2008) is one where the update is introduced using some mechanism that isn't considered as part of the system. An example of this would be the change of a physical piece of hardware, or making use of some system level code that isn't available to normal message passing programs. We use the notation $\psi_\emptyset$ to indicate an empty list of Predicated Updates. In order to make our analysis tractable we assume that a program starts with no pending updates, and that only one update can be introduced into the system. That update, however, can be introduced at any point in the execution of the original program. This approach could be extended to multiple updates by considering an updated program as the initial program for the second update, and applying the analyses presented in this thesis for single updates, on the modified code with a new update.

Normal reductions (Figure 3.5) can only occur when no updates are applicable:

$$\frac{\boxed{[\sigma]}\ P \xrightarrow{\gamma} \boxed{[\sigma']}\ P' \qquad \nexists p \in \mathsf{dom}(\psi).p(\sigma, P)}{\boxed{[\sigma]}\ P, \psi \xrightarrow{\gamma} \boxed{[\sigma']}\ P', \psi} \tag{4.8}$$

$$
\begin{array}{rcll}
\psi, \omega & ::= & & \textit{Predicated Updates} \\
& | & p \mapsto u & \text{Map from predicate to update} \\
& | & \psi, \psi & \text{List of predicate updates} \\
& | & \emptyset & \text{Empty predicate update} \\
\\
u & ::= & & \textit{Region updates} \\
& | & f \mapsto t & \text{Map from region to new code} \\
& | & u, u & \text{Set of updates} \\
& | & \emptyset & \text{Empty update} \\
\\
p & ::= & & \textit{Update predicate} \\
& | & p(z_{\Sigma}, z_{\Phi}, z_{\psi}) & \text{on code and message queues and current updates}
\end{array}
$$

Figure 4.14: Update Calculus

We define all the additions to the operational semantics that handle updates in Figure 4.10. We make use of the auxiliary function `regions`(t) that is inductively defined over the structure of terms, and returns a set of the region annotations used in t.

We provide a full reduction example for update in Figure 4.13. The region update maps regions f and g to the new region bodies t'_p and t'_{co} respectively. The update predicate states that the only valid point at which the update can be applied is when both the producer and the consumer are at the top of their recursive loops. The update is introduced on line `4`, but the update predicate doesn't become true until line `6`, at which point the update is applied on line `7`.

We assume that the new code included in a region update is all typable as Unit under an empty type environment:

$$\forall f \mapsto t \in u \,.\, \varphi \wr \emptyset \vdash t \colon \textsc{Unit} \tag{4.9}$$

There is a wide variety of work that considers how to update functions in a way that changes their type signature (Baumann et al., 2007; Soules et al., 2003; Baumann et al., 2005; Subramanian, 2010; Altekar et al., 2005; Kaashoek and Arnold, 2009). The techniques therewith are largely orthogonal to ours. We work under the simplifying assumption that the types of bodies of code will not change (in particular, that they have the unit assumption). This is as, to incorporate the techniques used to prove safety of changing type signatures as well, would have significantly complicated the presentation.

We refer to such a region update as *well formed*; we only consider well formed updates.

$$\frac{\varphi \wr \Gamma \vdash t : T}{f(\varphi) \wr \Gamma \vdash f(t) : T} \qquad \frac{\varphi \wr \Gamma \vdash t : T}{\texttt{dmod}_{\widetilde{f}}(\varphi) \wr \Gamma \vdash \texttt{dmod}_{\widetilde{f}}(t) : T}$$

Figure 4.15: Extensions To Typing Rules

$$\frac{}{[\Sigma]\ f(\epsilon) \xrightarrow{\tau} [\Sigma]\ \epsilon} \qquad \frac{[\Sigma]\ \varphi \xrightarrow{\tau}_F [\Sigma']\ \varphi'}{[\Sigma]\ f(\varphi) \xrightarrow{\tau}_F [\Sigma']\ f(\varphi')} \qquad \frac{[\Sigma]\ \varphi \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\ \varphi'}{[\Sigma]\ f(\varphi) \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\ \varphi'}$$

$$\frac{[\Sigma]\ \varphi_1 \xrightarrow{\gamma}_{F'} [\Sigma']\ \varphi_1' \qquad \texttt{regions}(\varphi_1) \not\subseteq F}{[\Sigma]\ \texttt{dmod}_F(\varphi_1) \xrightarrow{\gamma}_{F'} [\Sigma']\ \texttt{dmod}_{F\cup F'}(\varphi_1')} \qquad \frac{\texttt{regions}(\varphi_1) \subseteq F}{[\Sigma]\ \texttt{dmod}_F(\varphi_1) \xrightarrow{\tau} [\Sigma]\ \varphi_1}$$

$$\frac{[\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \qquad \nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi)}{[\Sigma]\ \Phi, \psi \xrightarrow{\gamma} [\Sigma']\ \Phi', \psi} \qquad \frac{\psi' \neq \emptyset \text{ introduced externally}}{[\Sigma]\ \Phi, \psi \xrightarrow{\tau} [\Sigma]\ \Phi, (\psi, \psi')}$$

$$\frac{\begin{array}{cc} \psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n & i \text{ smallest in } 1...n.p_i(\Sigma, \Phi) \\ \Phi = \varphi_1 \parallel \ldots \parallel \varphi_n & \varphi_j' = \mathsf{upd}(\varphi_j, u_i, \texttt{false}) \end{array}}{[\Sigma]\ \Phi, \psi \xrightarrow{\tau} [\Sigma]\ \varphi_1' \parallel \ldots \parallel \varphi_n', \psi \setminus p_i \mapsto u_i}$$

Figure 4.16: Update Extensions To Effect Semantics

4.2.3 Behavioural Abstraction

We extend the behavioural abstraction we defined in Section 3.3 to accommodate updates. We include typing rules for updatable regions and transactional regions (Figure 4.15). The effect reduction rules are extended to cover update (Figure 4.16); these mirror the concrete semantics. We abuse notation for the purposes of update predicates etc., assuming that:

$$\exists \Sigma, \Phi.p(\Sigma, \Phi) \Leftrightarrow \exists P, \sigma. \vdash P : \Phi \wedge \Sigma = \Lambda(\sigma) \wedge p(\sigma, P) \tag{4.10}$$

We define the update function over effects similarly to over terms (Figure 4.17). The most notable case is:

$$\mathsf{upd}(f(\varphi), u, b) \overset{\text{def}}{=} f(\varphi') \quad b \wedge f \in \mathsf{dom}(u) \wedge \varphi' \wr \emptyset \vdash u(f) : \_ \tag{4.11}$$

Here, instead of substituting the new body we instead determine its effect, under an empty type environment (which is possible for well formed updates), and substitute the effect.

4.3 General Formulation

In order to provide safety and liveness guarantees we must place restrictions on which programs are valid, in order to rule out errors and deadlock. We define a program point to consist of an effect Φ, a shared state abstraction Σ, and possibly an update. A program point is safe and deadlock free if it does not contain an error, if all its reductions result in safe and deadlock free program points, and if it can perform at least one reduction (or consists of empty effects). We formalise these in the following section. We then proceed to prove key properties of our update system.

4.3.1 Global Update Compatibility

In order to rule out deadlock and errors we must prove three things: 1) that whenever the underlying language reductions are enabled that no errors are present and that the system is not deadlocked, 2) that update reductions leave a program and messages in a compatible state, and 3) that introducing the update at any point in the execution is safe. We refer the property that comprises these conditions as *Global Update Compatibility* and formalise them function that is inductively defined in Figure 4.18; this is an extension of our earlier work (Anderson and Rathke, 2011).

The predicate $\mathcal{G}(\Sigma, \Phi, \psi, \omega)$ denotes that, given a program abstracted by Φ, shared state Σ, a list of pending updates ψ, and an update ω that can be introduced into the list of pending updates at any time, that no reductions will reduce in type errors, that the system is not deadlocked, and that any updates will leave the system in a safe state. We describe the restrictions in more detail below.

The predicate can alternatively be characterised as the greatest fixed point of the following function, defined over the state space S of tuples $(\Sigma, \Phi, \psi, \omega)$ of abstract states, parallel effects, and pairs of updates.

$$g(S) = \left\{ (\Sigma, \Phi, \psi, \omega) \;\middle|\; \begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \forall p \in \mathrm{dom}(\psi).\neg p(\Sigma, \Phi) \text{ and } \forall \Sigma', \Phi'.[\Sigma]\Phi \rightarrow [\Sigma']\Phi' \Rightarrow \\ (\Sigma', \Phi', \psi, \omega) \in S \text{ and } (\exists \Sigma', \Phi'.\, [\Sigma]\Phi \rightarrow [\Sigma']\Phi' \\ \qquad\qquad\qquad\qquad \text{or } \Phi \equiv \Pi\, \epsilon) \end{array} \right] \\ \text{or, for the first } p \mapsto u \text{ in } \psi \text{ such that } p(\Sigma, \Phi) \text{ we have} \\ \quad (\Sigma, \mathtt{upd}(\Phi, u), (\psi \setminus p \mapsto u), \omega) \in S \end{array} \right] \\ \text{and } \nexists \mathtt{error} \in \Phi \\ \text{and} (\Sigma, \Phi, (\psi, \omega), \emptyset) \in S \end{array} \right\} \tag{4.12}$$

This is a monotone function on the powerset lattice of tuples of abstract states, effects, updates, and updates. It has a greatest fixed point νg (which is just a set of tuples). Using the Knaster-Tarski fixed point theorem, we know that the predicate $\mathcal{G}(\Sigma, \Phi, \psi, \omega)$ is defined if and only if $(\Sigma, \Phi, \psi, \omega) \in \nu g$.

$$
\begin{array}{lll l}
\mathsf{upd}(\epsilon, u, b) & \stackrel{\text{def}}{=} & \epsilon & \\
\mathsf{upd}(\underline{X}, u, b) & \stackrel{\text{def}}{=} & \underline{X} & \\
\mathsf{upd}((\alpha(\widetilde{T}), T), u, b) & \stackrel{\text{def}}{=} & (\alpha(\widetilde{T}), T) & \\
\mathsf{upd}(\varphi_1;\ \varphi_2, u, b) & \stackrel{\text{def}}{=} & \mathsf{upd}(\varphi_1, u, b);\ \mathsf{upd}(\varphi_2, u, b) & \\
\mathsf{upd}(\varphi_1 \oplus \varphi_2, u, b) & \stackrel{\text{def}}{=} & \mathsf{upd}(\varphi_1, u, b) \oplus \mathsf{upd}(\varphi_2, u, b) & \\
\mathsf{upd}(\varphi_1 \& \varphi_2, u, b) & \stackrel{\text{def}}{=} & \mathsf{upd}(\varphi_1, u, b) \& \mathsf{upd}(\varphi_2, u, b) & \\
\mathsf{upd}(\mu\underline{X}.\varphi, u, b) & \stackrel{\text{def}}{=} & \mu\underline{X}.\mathsf{upd}(\varphi, u, b) & \\
\mathsf{upd}(\mathtt{error}, u, b) & \stackrel{\text{def}}{=} & \mathtt{error} & \\
\mathsf{upd}(f(\varphi), u, b) & \stackrel{\text{def}}{=} & f(\mathsf{upd}(\varphi, u, b)) & (\neg b \vee f \mapsto\notin u) \\
\mathsf{upd}(f(\varphi), u, \mathtt{true}) & \stackrel{\text{def}}{=} & f(\varphi') & \nexists t. f \mapsto t \in u \wedge \\
 & & & \quad \varphi' \wr \emptyset \vdash t' \colon \textsc{Unit} \\
\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi), u, b) & \stackrel{\text{def}}{=} & \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi, u, \mathtt{true})) & \gamma \cap \mathsf{dom}(u) = \emptyset \\
\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi), u, b) & \stackrel{\text{def}}{=} & \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi, u, \mathtt{false})) & \gamma \cap \mathsf{dom}(u) \neq \emptyset
\end{array}
$$

Figure 4.17: Update Function On Effects

Language Reductions

An error occurs when a thread receives a value it does not expect. In order to rule out such cases we need to ensure that, at every point which a program can reach, if the program can perform a reduction it reduces to another valid state:

$$[\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \Rightarrow \mathcal{G}(\Sigma', \Phi', \psi, \omega) \tag{4.13}$$

Deadlock occurs when a process wants to access the shared state but the action is blocked. In order to rule out deadlock we need to ensure, at every point $[\Sigma]\ \Phi$ which a program can reach, that either it is able to perform a reduction, or it consists of empty effects:

$$[\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \vee \Phi = \Pi_I\, \epsilon \tag{4.14}$$

We can only perform reductions on the Message Passing Calculus if there are no updates which are applicable at that point:

$$\frac{[\sigma]\ P \xrightarrow{\gamma} [\sigma']\ P' \qquad \nexists p \in \mathsf{dom}(\psi).p(\sigma, P)}{[\sigma]\ P, \psi \xrightarrow{\gamma} [\sigma']\ P', \psi} \tag{4.15}$$

Hence we only need to ensure that (4.13) and (4.14) hold in situations where $\nexists p \in \mathsf{dom}(\psi).p(\Sigma, P)$:

$$\begin{aligned} &\nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi)\ \wedge\ (\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \vee \Phi \equiv \Pi_I\, \epsilon) \wedge \\ &\quad (\forall \Phi'.\ [\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \Rightarrow \mathcal{G}^k(\Sigma', \Phi', \psi, \omega)) \end{aligned} \tag{4.16}$$

$$\mathcal{G}^0(\Sigma,\Phi,\psi,\omega) \overset{\text{def}}{=} \texttt{true}$$

$$\mathcal{G}^k(\Sigma,\Phi,\psi,\omega) \overset{\text{def}}{=}$$
$$\left[\begin{array}{l} \left[\begin{array}{l} \nexists p \in \mathsf{dom}(\psi).p(\Sigma,\Phi)\wedge \\ \qquad (\forall \Phi'.\ [\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \Rightarrow \mathcal{G}^{k-1}(\Sigma',\Phi',\psi,\omega)) \\ \qquad \wedge (\exists\Sigma',\Phi'.\ [\Sigma]\ \Phi \xrightarrow{\gamma} [\Sigma']\ \Phi' \vee \Phi = \Pi_I\,\epsilon) \\ \vee \psi = p_1 \mapsto u_1,\ldots,p_n\mapsto u_n \wedge i \text{ smallest in 1...n.} \\ \qquad p_i(\Sigma,\Phi) \wedge \mathcal{G}^{k-1}(\Sigma, \mathsf{upd}(\Phi,u_i,\texttt{false}), \psi \setminus p_i \mapsto u_i, \omega) \end{array}\right] \\ \wedge \nexists \texttt{error} \in \Phi \wedge \mathcal{G}^{k-1}(\Sigma,\Phi,(\psi,\omega),\emptyset) \end{array}\right.$$
where $k > 0$

$$\mathcal{G}(\Sigma,\Phi,\psi,\omega) = \bigwedge_{k\in\mathcal{N}} \mathcal{G}^k(\Sigma,\Phi,\psi,\omega)$$

Figure 4.18: Global Update Compatibility

4.3.1.1 Applying Updates

If a pending Predicated Update is applicable (if its update predicate is true) we apply it straight away. Hence we require, whenever a pending Predicate Update is applicable, that it leaves the system in a Globally Consistent state:

$$\begin{array}{l} \psi = p_1 \mapsto u_1,\ldots,p_n \mapsto u_n \wedge i \text{ smallest in } 1\ldots n. \\ \qquad p_i(\Sigma,\Phi) \wedge \mathcal{G}(\Sigma, \mathsf{upd}(\Phi,u_i,\texttt{false}), (\psi \setminus p_i \mapsto u_i), \omega) \end{array} \tag{4.17}$$

4.3.1.2 Introducing Updates

Updates can be introduced into the system at any time:

$$\frac{}{[\sigma]\ P,\psi \xrightarrow{\tau} [\sigma]\ P,(\psi,\omega)} \tag{4.18}$$

In order to make our work tractable we assume that a program commences execution with no pending updates, and that only one update, ω, is introduced in the entire course of the program. We require that if we move the update we are considering into the list of pending updates, that the system remains Globally Compatible:

$$\mathcal{G}(\Sigma,\Phi,(\psi,\omega),\emptyset) \tag{4.19}$$

4.3.2 Key Properties

As discussed in Section 3.4, verifying that Global Update Compatibility holds for a given system amounts to model checking, and such a check is computationally expensive.

We again permit stronger properties which imply General Compatibility for updatable programs:

$$\frac{\vdash P\colon \Phi \qquad \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma), \Phi, \psi, \omega) \text{ and } (\mathcal{C} \implies \mathcal{G})}{\vdash_\omega \boxed{[\sigma]}\ P, \psi} \tag{4.20}$$

The judgement $\vdash_\omega \boxed{[\sigma]}\ P, \psi$ defines that a program P, with shared state σ, pending updates ψ, and an update which can become pending at some arbitrary point in the future ω, has Global Update Compatibility. We refer to program points that can be validated using the rule in Equation 4.20 to be *update valid*.

We can prove that an update valid program points will retain safety in the presence of update.

Theorem 4.1. *Subject Reduction of Update Valid Program Points*

For all $\omega, \sigma, P, \psi, \gamma, \sigma', P', \psi'$ if:

$$\vdash_\omega \boxed{[\sigma]}\ P, \psi \qquad \boxed{[\sigma]}\ P, \psi \xrightarrow{\gamma} \boxed{[\sigma']}\ P', \psi'$$

then there exists ω' such that

$$\vdash_{\omega'} \boxed{[\sigma']}\ P', \psi'$$

Proof. Suppose the hypotheses. We then prove the conclusion by induction over the derivation tree of the reduction $\boxed{[\sigma]}\ P, \psi \xrightarrow{\gamma} \boxed{[\sigma']}\ P', \psi'$.

Consider the case where the last reduction rule used is RNoUpd:

$$\frac{\boxed{[\sigma]}\ P \xrightarrow{\gamma}_F \boxed{[\sigma']}\ P' \qquad \nexists p \in \mathsf{dom}(\psi).p(\sigma, P)}{\boxed{[\sigma]}\ P, \psi \xrightarrow{\gamma} \boxed{[\sigma']}\ P', \psi}$$

By the reduction rule we know that $\nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi)$. We can use Lemma 3.2 to show that as $\boxed{[\sigma]}\ P \xrightarrow{\gamma} \boxed{[\sigma']}\ P'$ then we know that $\vdash P'\colon \Phi'$ and $\boxed{[\Lambda(\sigma)]}\ \Phi, \psi \xrightarrow{\gamma'} \boxed{[\Lambda(\sigma')]}\ \Phi', \psi'$. Then, by the definition of $\mathcal{G}$, we can know that: $\nexists\texttt{error} \in \Phi$ and $\boxed{[\Sigma]}\ \Phi \rightarrow \boxed{[\Sigma']}\ \Phi' \Rightarrow \mathcal{G}(\Sigma', \Phi', \psi, \omega)$ and hence that $\nexists\texttt{error} \in \Phi'$. Finally, given that $\mathcal{G}(\Sigma', \Phi', \psi, \omega)$, then letting $\mathcal{C} = \mathcal{G}$, we can show that: $\vdash_\omega \boxed{[\sigma']}\ P', \psi$, using the rule in Equation 4.20.

Consider the case where the last reduction rule used is RIntrUpd. By the definition of $\mathcal{G}$ we know that $\mathcal{G}(\Sigma, \Phi, (\psi, \omega), \emptyset)$. Hence we can straightforwardly show that $\vdash_\emptyset \boxed{[\sigma]}\ P, \psi, \omega$, using the rule in Equation 4.20.

Consider the case where the last reduction rule used is RUpd:

$$\frac{\psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n \qquad i \text{ smallest in } 1 \ldots \text{n}.p_i(\sigma, P)}{\boxed{[\sigma]}\ P, \psi \xrightarrow{\mathcal{T}} \boxed{[\sigma]}\ \mathtt{upd}(P, u_i, \mathtt{false}), \psi \setminus p_i \mapsto u_i}$$

By Lemma A.5 we can show that as $\vdash P \colon \Phi$ and $\forall f \mapsto t.\varphi \wr \emptyset \vdash t \colon \textsc{Unit}$ we know that $\vdash \mathtt{upd}(P, u, \mathtt{false}) \colon \mathtt{upd}(\Phi, u, \mathtt{false})$. By the definition of $\mathcal{G}$ we know that given that $\psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n$ there exists a smallest i such that $p_i(\Sigma, \Phi)$ holds, and furthermore that $\mathcal{G}(\Sigma, \mathtt{upd}(\Phi, u_i, \mathtt{false}), \psi \setminus p_i \mapsto u_i, \omega)$. Hence we can show that $\vdash_\omega [\sigma'] \ P', \psi$, using the rule in Equation 4.20. $\square$

We can prove that an update valid program points will retain liveness in the presence of update.

Theorem 4.2. *Liveness of Update Valid Program Points*

For all ω, σ, P, ψ, if:

$$\vdash_\omega [\sigma] \ P, \psi$$

then either $P = \Pi v$, or there exists $\gamma, \sigma', P', \psi'$ such that

$$[\sigma] \ P, \psi \xrightarrow{\gamma} [\sigma'] \ P', \psi'$$

Proof. It is the case that either there exists an update predicate that is fulfilled by the current shared state and program point, or there is not. We perform a case split on these conditions.

Consider the case where there is no applicable update. As $\vdash_\omega [\sigma] \ P, \psi$ holds then by the definition of $\mathcal{G}$, we know that either there exists Σ', Φ' such that $[\Sigma] \ \Phi \xrightarrow{\gamma} [\Sigma'] \ \Phi'$, or that $\Phi \equiv \Pi_I \, \epsilon$. We perform a case split between these two possibilities.

Consider the case where $[\Sigma] \ \Phi \xrightarrow{\gamma} [\Sigma'] \ \Phi'$. By Lemma 3.6 we know that $[\sigma] \ P \xrightarrow{\gamma} [\sigma'] \ P'$, and hence, using the (RNoUpd) rule we can show that $[\sigma] \ P, \psi \xrightarrow{\gamma} [\sigma'] \ P', \psi'$, as required.

Consider the case where $\Phi = \Pi_I \, \epsilon$. By Lemma 3.5 we know that either $[\sigma] \ P \xrightarrow{\gamma} [\sigma'] \ P'$ or $P = \Pi_I \, v$, as required.

Consider the case where there is an applicable update. We know that $\mathcal{G}$ holds. As there exists an update predicate that is true for the current program point we can choose the smallest i such that $p_i(\Sigma, \Phi)$ holds. Hence, using the (RNoUpd) rule, we can show that $[\sigma] \ P, \psi \xrightarrow{\gamma} [\sigma'] \ \mathtt{upd}(P, u, \mathtt{false}), \psi \setminus p_i \mapsto u_i$, as required $\square$

4.4 Conclusions

In this chapter we consider communication safety and liveness for side effecting programs that can be updated at runtime. Our contributions are as follows. We demonstrate several examples of how update can introduce errors and deadlock into otherwise safe and live programs. In particular we show how if a consumer is updated before a producer, in a producer consumer system with acknowledgement, deadlock can be introduced into the system. We show how to extend general compatibility to incorporate dynamic programs. This approach is of even higher complexity than that for global compatibility presented in Chapter 3. This is as each at each reachable program point obtained by interleaving of the program, we must check to see if it is safe to introduce an update, and that all interleavings of the updated program are also safe. We again rely on the implication that we will develop specific systems that have reduced verification complexity. These are explored in the next chapter.

Chapter 5

Specific Formulations for DSU

The cost of verifying general compatibility is un-necessarily expensive, due to the high checking complexity. We explore stronger properties that imply Global Update Compatibility, as these properties are often easier to prove, and reduce the cost of verification. In this chapter we present two techniques to statically prove safety of updates to message passing programs, where we informally argue that the cost of verification is linear in the modified program. The first technique is a form of Global Synchronisation that is straightforward to reason about and prove safe (Section 5.1). The second technique uses the semantics of blocking message passing to provide synchronisation so that we can update groups of threads separately rather than all threads at a synchronised point (Section 5.2). We contrast their trade-off between simplicity and restrictions on the programs and updates allowed, present our conclusions, and describe our novel contributions in Section 5.3.

$$G \ ::= \ d_1 \to d_2 : \langle \widetilde{T \mapsto G} \rangle \mid \mu \underline{X}.G \mid \underline{X} \mid \mathsf{0} \mid \colorbox{lightgray}{$f(G)$} \mid \colorbox{lightgray}{$\mathtt{dmod}(G)$}$$

Figure 5.1: Global Session Types With Update Extensions

$$f(G) \upharpoonright d \ \stackrel{\text{def}}{=} \ f_d(G \upharpoonright d) \qquad \mathtt{dmod}_F(G) \upharpoonright d \ \stackrel{\text{def}}{=} \ \mathtt{dmod}_{\{f_d \mid f \in F\}}(G \upharpoonright d)$$

Figure 5.2: Extended Global Session Type Projection Function

5.1 Global Typability

We extend Global Session Types to include updatable and transactional regions (Figure 5.1). The programs that most benefit from DSU are those that are structured as long running processing loops (Subramanian, 2010). Hence we consider pre-update message passing programs which can be abstracted using Global Session Types of the form $\mu\underline{X}.\mathtt{dmod}(f(G))$, where there are no additional region annotations in G; this denotes a long running loop that can be updated by replacing the all future iterations of a processing loop with another processing loop.

We define the projection of the extended Global Session Types in Figure 5.2 (extending the function in Figure 2.10). We handle projection of updatable and transactional regions as follows. The projection of annotation $f(G)$ onto a participant d is defined to be $f_d(G \upharpoonright d)$. Note the subscript d on the region name for the local Session Type of d; this simply permits us to ensure the region annotation in the Global Session Type is projected to syntactically separate region annotations in each of the participant's local Session Types. We do this in order to be able to update each process independently. The projection of transactional regions is straightforward.

5.1.1 Formalisation

One important feature of the Global Session Type approach is that each communication action in the Global Type projects to a *pair* of send/receive actions in the local Session Types. As this pair of actions is decoupled into independent processes, in terms of protocol a point where one of the actions has been executed and the other hasn't is not a suitable point for update. Checking for emptiness of queues would be sufficient to rule out this case, however this Global action may be a part of a larger transaction, as indicated by the regions. In order to maintain transactional sanity we do not want to permit one process to be updated when another is not, as in Figure 4.3. As processes can enter the transactional regions in the local types independently, we must also verify that either all processes involved in that particular transactions have entered the region, or all processes have not. A straightforward way of doing this is to check that the processes

$$\Delta^{\restriction G} \stackrel{\text{def}}{=} \bigcup_{n<\omega} \Delta_n^{\restriction G} \qquad \Delta_0^{\restriction G} \stackrel{\text{def}}{=} \{\, [\Sigma_\emptyset]\ \Phi^{\restriction G}\}$$

$$\begin{aligned}
\Delta_{n+1}^{\restriction G} \quad \stackrel{\text{def}}{=} \quad & \{\, \delta(\,[c \mapsto T;\, \Sigma^k]\,, \varphi_b', \varphi_a^k) \,|\, \forall j \in J.\delta(\Sigma^j, \varphi_b^j, \varphi_a^j) \in \Delta_n^{\restriction G})\} \\
& \cup \{\, \delta(\,[\Sigma]\,, \varphi_a', \varphi_b^j) \,|\, \forall k \in K\,.\,\delta(\Sigma, \varphi_a^k, \varphi_b^k) \in \Delta_n^{\restriction G} \ \wedge\ \Sigma(c) = \emptyset) \\
& \cup \{\, \delta(\,[\Sigma]\,, \mathtt{dmod}_F(\varphi)) \,|\, \delta(\Sigma, \varphi) \in \Delta_n^{\restriction G} \ \wedge\ F \subseteq \mathtt{regions}(\varphi)) \\
& \cup\ \Delta_n^{\restriction G}
\end{aligned}$$

where $c = d_a d_b, \emptyset \subset K \subseteq J$, all G are of the form $\mathsf{dmod}(f(G'))$, that the only place that the $\mathtt{dmod}()$ and $f()$ annotations appear is at top level, and

$$\varphi_a' = \begin{cases} \mathsf{dmod}_\emptyset(f(\oplus_K(c!(T_k);\ \varphi_a^k))) & \text{if } G \restriction d_a \equiv \mathsf{dmod}_\emptyset(f_{d_i}(\oplus_K(c!(T_k);\ \varphi_a^k))) \\ \oplus_K(c!(T_k);\ \varphi_a^k & \text{otherwise} \end{cases}$$

$$\varphi_b' = \begin{cases} \mathsf{dmod}_\emptyset(f(\&_J(c?(T_j);\ \varphi_b^j)))) & \text{if } G \restriction d_b \equiv \mathsf{dmod}_\emptyset(f_{d_i}(\&_J(c?(T_j);\ \varphi_b^j)))) \\ \&_J(c?(T_j);\ \varphi_b^j) & \text{otherwise} \end{cases}$$

Figure 5.3: Annotated Blocking Message Passing Safe Program Points Set

are typable using *some* Global Session Type, and that the message queues are empty. This guarantees that communications actions are in parity, and also that processes are in consistent states with respect to transactions. We call this approach *Global Typability*, and formalise its constraints on the updates that are permissible as:

$$\mathcal{GT}(\psi) \text{ iff } (\psi = p_{gt}(z_\Sigma, z_\Phi, z_\psi) \mapsto \{\widetilde{f_{d_i} \mapsto t_{d_i}}\} \text{ and } \exists \Phi.\vdash \prod t_{d_i} : \Phi^{\restriction}) \qquad (5.1)$$

where $p_{gt}(z_\Sigma, z_\Phi, z_\psi)$ is the predicate $(z_\Sigma = \Sigma_\emptyset)\ \wedge\ z_{\Phi^{\restriction}}$, and $z_{\Phi^{\restriction}}$ denotes that z_Φ is the projection of some Global Session Type.

In order to ensure that updates transform programs from (provably) safe and live states to (provably) safe and live states, we must extend the consistency definition from Figure 3.13 to account for possible program states that can be reached by reducing a program with $\mathtt{dmod}()$ and $f()$ annotations. We modify the inductive definition of a safe program points sets to comprise such points (Figure 5.3). Note that, unlike in Figure 3.13, the safe points set is parameterised by a global session type G. This is used to maintain correct encapsulation with respect to updateable and transactional regions. We return to this in the description of the second construction, below. The base case consists of program points that are projections of Global Session Types, with accompanying empty message queues. Here we make explicit our assumption that programs are of the form $\mu\underline{X}.\mathtt{dmod}(f(G_1))$, and that the only place that the $\mathtt{dmod}()$ and $f()$ annotations appear is at top level. In addition, we require that the body of the protocol only appear at top level, under the recursive binding. We will return to this requirement momentarily.

$$
\begin{array}{ll}
1 & [\sigma_\emptyset]\ (t_c \parallel t_s), \psi_\emptyset \\
2 & \quad \rightarrow \ldots \\
3 & \quad \rightarrow [c_1 \mapsto \texttt{prime?}]\ (\texttt{snd}(c, 3457).\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
  & \qquad \mu \underline{X}'.g(\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \ldots\})), \psi_\emptyset \\
4 & \quad \rightarrow [c_1 \mapsto \texttt{prime?}]\ (\texttt{snd}(c, 3457).\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
  & \qquad \mu \underline{X}'.g(\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \ldots\})), \psi \\
5 & \quad \rightarrow [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
  & \qquad \mu \underline{X}'.g(\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).\underline{X}', \ldots\})), \psi \\
6 & \quad \rightarrow [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
  & \qquad g(\texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t_s, \ldots\})), \psi \\
7 & \quad \rightarrow [c_1 \mapsto \texttt{prime?}, 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \\
  & \qquad \texttt{case}\ c_1\ \texttt{of}\ \{\texttt{prime?} \mapsto \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t_s, \ldots\}), \psi \\
8 & \quad \rightarrow [c_1 \mapsto 3457]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \texttt{rcv}(c_1)(x : \textsc{Int}).\texttt{snd}(c_2, \ldots).t_s), \psi \\
9 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t_c \parallel \texttt{snd}(c_2, \ldots).t_s), \psi \\
10 & \quad \rightarrow [\sigma_\emptyset]\ (\texttt{rcv}(c_2)(x : \textsc{Int}).t'_c \parallel \texttt{snd}(c_2, \ldots).t'_s), \psi_\emptyset \\
11 & \quad \rightarrow \ldots
\end{array}
$$

where

$$\psi = p \mapsto (f \mapsto t'_c, g \mapsto t'_s) \qquad p_s(\Sigma, \Phi) \overset{\text{def}}{=} (\Sigma = \Sigma_\emptyset \wedge \exists G.\Phi = G \upharpoonright d_1 \parallel \ldots \parallel G \upharpoonright d_n)$$

Figure 5.4: Update Success In Maths Server Using Global Typability

The first and second constructions are similar to those for blocking message passing in Figure 3.13, in that they add a value to the front of the queue with a complementary external choice on receive, and add complementary internal choice between sends and external choice between receives, respectively. There is an additional aspect to the construction, however, that we explain by example. We want to guarantee that all threads are updated, and that an update does not result in old code intreating with new code. Specifically, we wish to rule out situations such as:

$$[\Sigma_\emptyset]\ \mu\underline{X}.\texttt{dmod}_\emptyset(f(c!\langle T\rangle;\ \underline{X})) \parallel \mu\underline{X}.(c?(T);\ \underline{X}) \tag{5.2}$$

In this case we could update the producer to be empty, thus introducing deadlock into the system. Hence we require that any time we use a construction rule, that if the extended process is the projection of the top level protocol $\mu\underline{X}.\texttt{dmod}(f(G_1))$, that the relevant $\texttt{dmod}()$ and $f()$ annotations must be added. The third construction permits us to add a $\texttt{dmod}_F()$ where the region set F comprises all regions within the enclosed process; this means that the next reduction for this process will simply strip away the extraneous annotation.

We can use Global Typability to perform the update to the maths server (from Figure 4.1 to Figure 4.2) using the update predicate for Global Typability. An example reduction for such an update is shown in Figure 5.4. On line `4` we introduce the update when there are values in the message queues. The program then continues evaluating until the update predicate becomes true on line `9`, after which the update is performed on line `10`. The Global Update predicate will rule out example erroneous reductions such as in Figures 4.3.

5.1.2 Properties and Proofs

In order to prove that Global Typability implies Global Compatibility we make use of the following lemmas. First we show that any program point within the set of safe program points reduces, then the new program point is still within the set of safe program points.

Lemma 5.1. *Valid Simple Points Reduction*

$$\forall \Sigma, \Phi, G_1, n, \Sigma', \Phi'.\ \boxed{[\Sigma]}\ \Phi \in \Delta_n^{\upharpoonright G_1} \wedge \boxed{[\Sigma]}\ \Phi \rightarrow \boxed{[\Sigma']}\ \Phi' \Rightarrow \boxed{[\Sigma']}\ \Phi' \in \Delta_n^{\upharpoonright G_1}$$

Proof. The proof proceeds similarly to that in Lemma 3.9. Suppose the hypotheses. We then prove the conclusion by induction over the n of $\Delta_n^{\upharpoonright G_1}$. The key differences are in the case where $n > 0$. By the structure of parallel effects we can identify one effect which is the one that reduces:

$$\boxed{[\Sigma]}\ \varphi_1 \parallel \ldots \parallel \varphi_i \parallel \ldots \parallel \varphi_m \rightarrow \boxed{[\Sigma']}\ \varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \varphi_m$$

As $\boxed{[\Sigma]}\ \Phi \in \Delta_n^{\upharpoonright G_1}$ we know that $\boxed{[\Sigma]}\ \Phi$ is either in one of the sets included in the definition of $\Delta_n^{\upharpoonright G_1}$, or it is in $\Delta_{n-1}^{\upharpoonright G_1}$.

Consider the case when $\boxed{[\Sigma]}\ \Phi \in \{\delta([c \mapsto T; \Sigma^k], \varphi_b')\}$ where $\forall j \in J. \delta(\Sigma^j, \varphi_b^j) \in \Delta_{n-1}^{\upharpoonright G_1}$ (as in Figure 5.3). We can perform a case split over the index i. If $i = b$ then φ_b is of one of two forms. If $\varphi \equiv \mathsf{dmod}_\emptyset(f(\&_J(c!(T_j);\ \varphi_b^j))))$ then by the reduction rules, we know that $\varphi_i' \equiv \mathsf{dmod}_{f_{d_b}}(\varphi_b^k)$ and that $\Sigma' = \Sigma_k$. As we have that $\delta(\Sigma^k, \varphi_a^k, \varphi_b^k) \in \Delta_{n-1}^{\upharpoonright G_1}$, then by the third construction in Figure 5.3 we have that $\delta(\Sigma^k, \mathsf{dmod}_{f_{d_b}}(\varphi_b^k)) \in \Delta_{n-1}^{\upharpoonright G_1}$, as required. The cases where $\varphi \equiv \&_J(c!(T_j);\ \varphi_b^j)$ and $i \neq b$ continue as in Lemma 3.9.

Consider the case when $\boxed{[\Sigma]}\ \Phi \in \{\ \delta(\Sigma, \varphi_a', \varphi_b')\}$ where $\forall j \in J. \delta(\Sigma^j, \varphi_a^j, \varphi_b^j) \in \Delta_{n-1}^{\upharpoonright G_1}$ and $\Sigma(c) = \emptyset$ (as in Figure 5.3). This follows the first case, and has four permutations (rather than two), as φ_a' can be of form $\mathsf{dmod}_\emptyset(f(\oplus_K(c!(T_k);\ \varphi_a^k)))$ and of form $\oplus_K(c!(T_k);\ \varphi_a^k)$, and $\oplus_K(c!(T_k);\ \varphi_a^k)$ can either be a true choice, or a degenerate choice $c!(T_k);\ \varphi_a^k$, again making use of the third construction in Figure 5.3 when $\varphi_a' \equiv \mathsf{dmod}_\emptyset(f(\oplus_K(c!(T_k);\ \varphi_a^k)))$.

Consider the case when $\boxed{[\Sigma]}\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_F(\varphi_i)\}$ where $\delta(\Sigma, \varphi_i) \in \Delta_{n-1}^{\upharpoonright G_1}$ and $F \subseteq \mathsf{regions}(\varphi_a)$ (as in Figure 5.3). By the (DMod) rule we know that $\boxed{[\Sigma]}\ \varphi_i \rightarrow \boxed{[\Sigma']}\ \varphi_i'$. As $\delta(\Sigma, \varphi_i) \in \Delta_{n-1}^{\upharpoonright G_1}$, then by the inductive hypothesis we can show that $\delta(\Sigma', \varphi_i') \in$

$\Delta_{n-1}^{\restriction G_1}$. Then by the third construction we can show that $\delta(\Sigma', \mathsf{dmod}_F(\varphi_i')) \in \Delta_{n-1}^{\restriction G_1}$, as required □

We can prove that valid program points are safe and live.

Lemma 5.2. *Valid States Safety Lemma*

$$\forall \Sigma, \Phi, G, n.\ \boxed{\Sigma}\ \Phi \in \Delta_n^{\restriction G} \Rightarrow \mathit{error} \notin \Phi$$

Proof. Similar to that of Lemma 3.10. □

Lemma 5.3. *Valid States Liveness*

$$\forall \Sigma, \Phi, G, n.\ \boxed{\Sigma}\ \Phi \in \Delta_n^{\restriction G} \Rightarrow \exists \Sigma', \Phi'.\ \boxed{\Sigma}\ \Phi \rightarrow \boxed{\Sigma'}\ \Phi' \vee \Phi \equiv \Pi\, \epsilon$$

Proof. The proof proceeds similarly to that of Lemma 3.11, by induction over n. The key difference is where $n > 0$ and where $\boxed{\Sigma}\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_F(\varphi_i)\}$, given that $\delta(\Sigma, \varphi_i) \in \Delta_{n-1}^{\restriction G}$ and $F \subseteq \mathsf{regions}(\varphi)$. By the reduction rules, as $F \subseteq \mathsf{regions}(\varphi)$, we know that $\boxed{\Sigma}\ \mathsf{dmod}_F(\phi) \rightarrow \boxed{\Sigma}\ \phi$, and hence

$$\boxed{\Sigma}\ \phi_1 || \ldots || \mathsf{dmod}_F(\phi) || \ldots || \phi_m \rightarrow \boxed{\Sigma}\ \phi_1 || \ldots || \phi || \ldots || \phi_m$$

as required. □

We prove that, if we apply a well formed update to a valid (abstract) process point, that the updated point is still valid.

Lemma 5.4. *Global Session Type Updatability*

$\forall \Phi, G, u, f_i, t_i, \phi_i, G', b.$ *if*

$$\Phi \equiv \Phi^{\restriction G} \qquad u = \widetilde{f_i \mapsto t_i}_I \qquad \varphi_i \wr \emptyset \vdash t_i \colon \textsc{Unit} \qquad G' \restriction d_i \equiv \varphi_i$$

Then $\exists G'', \Phi'$ *such that*

$$\mathsf{upd}(\Phi, u, b) \equiv \Phi'^{\restriction G''}$$

Proof. Suppose the hypotheses. We then prove the conclusion by induction over the structure of G. The key cases are as follows.

Consider the case where $G = f(G_1)$. The update function depends on the boolean value b. Hence we perform a case split on its possible values. Consider the case $b = \mathtt{true}$. By the definition of $\upharpoonright$ we know that $\Pi\, f(G_1) \upharpoonright d \equiv \Pi\, f_d(G_1 \upharpoonright d)$. By Lemma A.4 and the definition of `upd` we know that:

$$\begin{array}{rcl} \Pi\,\mathtt{upd}(f(G_1) \upharpoonright d, u, \mathtt{true}) & \equiv & \Pi\,\mathtt{upd}(\varphi'_i, u, \mathtt{true}) \\ & \equiv & \Pi\,\mathtt{upd}(G' \upharpoonright d, u, \mathtt{true}) \end{array}$$

as required. Consider the case where $b = \mathtt{false}$. By Lemma A.4 and the definition of `upd` we know that $\Pi\,\mathtt{upd}(f(G_1) \upharpoonright d, u, \mathtt{false}) \equiv \Pi\, f_d(\mathtt{upd}(G_1 \upharpoonright d, u, \mathtt{false}))$. We can use the inductive hypothesis to show that $\Pi\,\mathtt{upd}(G_1 \upharpoonright d, u, \mathtt{false}) \equiv \Pi\, G''' \upharpoonright d$. Hence we have that $\Pi\,\mathtt{upd}(f(G_1) \upharpoonright d, u, \mathtt{false}) \equiv \Pi\, f_d(\mathtt{upd}(G''' \upharpoonright d, u, \mathtt{false}))$ and $G'' = f(G''')$, as required.

Consider the case where $G = \mathtt{dmod}_F(G_1)$. By Lemma A.4 and the definition of `upd` we know that $\Pi\,\mathtt{upd}(\mathtt{dmod}_F(G_1) \upharpoonright d, u, b) \equiv \Pi\,\mathtt{dmod}_{F_d}(\mathtt{upd}(G_1 \upharpoonright d, u, \mathtt{true}))$, where $\mathtt{regions}(G_1) \cap F = \emptyset$ and $\Pi\,\mathtt{upd}(\mathtt{dmod}_F(G_1) \upharpoonright d, u, b) \equiv \Pi\,\mathtt{dmod}_{F_d}(\mathtt{upd}(G_1 \upharpoonright d, u, \mathtt{false}))$, where $\mathtt{regions}(G_1) \cap F \neq \emptyset$. By the inductive hypothesis we can show that $\Pi\,\mathtt{upd}(G_1 \upharpoonright d, u, b) \equiv \Pi\, G''' \upharpoonright d$. Hence we have that $\Pi\,\mathtt{upd}(\mathtt{dmod}_F(G_1) \upharpoonright d, u, b) \equiv \Pi\,(\mathtt{dmod}_F(G''')) \upharpoonright d$, as required. □

We can show that if we apply a well formed update to a valid program point then the updated point is still valid.

Lemma 5.5. *Global Typability Update Consistency*

If:

$$\vdash P : \Phi \qquad \exists G.\Phi = G \upharpoonright d_1 \parallel \ldots \parallel G \upharpoonright d_n$$
$$u = f_{d_1} \mapsto t_{d_1}, \ldots, f_{d_n} \mapsto t_{d_n}.\exists G'.\forall i \in 1 \ldots n\,.\, G \upharpoonright d_i \wr \emptyset \vdash t_{d_i} : T$$

then:

$$\vdash \mathtt{upd}(P, u, \mathit{false}) : \Phi' \qquad \exists G'.\Phi' = G' \upharpoonright d_1 \parallel \ldots \parallel G' \upharpoonright d_n$$

Proof. By induction over the structure of G. □

We can then prove that if such an update is introduced into a (valid) message passing program that has reduced from the projection of a Global Session Type, that whenever the update is applied it will result in another valid program. An informal explanation of this proof is as follows. Up until the point that an update is introduced, and its update predicate has become true, we can use the fact that the program point is a member of the valid points set to show that the program is live and safe, as in Observation 3.13 and Observation 3.14. At the point the update is applied we know that the message queues are empty, and the effect of the process is the projection of a Global Session Type. When the well formed update is applied to the program we can use Lemma 5.4 to show that the updated program's effect is still the projection of a Global Session Type. As the message queues are still empty then we can show that $\vdash \mathtt{upd}(P, u, \mathtt{false}) : \Phi'^{\upharpoonright}$.

and hence that $\boxed{\Sigma}\ \mathtt{upd}(\Phi, u, \mathtt{false}) \in \Delta_0^{G_2}$. After the update is applied we know that no updates will be introduced or applied again. hence we can show that the resulting program is safe and live, as in Section 3.5.1.

Theorem 5.6. *Global Typability Theorem*

For all $k, \Sigma, P, \psi, \omega$, if ($\mathcal{GT}(\psi)$ or $\phi = \psi = \emptyset$) and $\boxed{\Sigma}\ \Phi \in \Delta^{G_1}$ then $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$.

Proof. Suppose the hypotheses. We then prove the conclusion by induction over the k in $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$.

Consider the case where $k = 0$. We trivially have that $\mathcal{G}^0(\Sigma, \Phi, \psi, \omega) = \mathtt{true}$.

Consider the case where $k > 0$. We prove each of the conjuncts of $\mathcal{G}$ separately. In order to prove the second conjunct: by Lemma 5.2 we can show that $\nexists \mathtt{error} \in \Phi'$, as required. We can show the third conjunct directly using the inductive hypothesis, and hence have $\mathcal{G}^{k-1}(\Sigma, \Phi, (\psi, \omega), \emptyset)$ as required, as $k - 1 < k$. In order to prove the third conjunct we must either show that no update is applicable, and for all possible reductions the resulting program points are globally compatible, or if an update is applicable then the updated program point is globally compatible.

Consider the case where there are no applicable updates. By Lemma 5.1 we know that if there exists Σ' and Φ' such that there is a reduction $\boxed{\Sigma}\ \Phi \rightarrow \boxed{\Sigma'}\ \Phi'$ then the resulting program point $\boxed{\Sigma'}\ \Phi'$ are still in the valid points set, $\boxed{\Sigma'}\ \Phi' \in \Delta^{G_1}$. By the inductive hypothesis, as $\mathcal{GT}(\psi)$ and $\boxed{\Sigma'}\ \Phi' \in \Delta^{G_1}$ then we know that $\mathcal{G}^{k-1}(\Sigma', \Phi', \psi, \omega)$. By Lemma 5.3, then as $\boxed{\Sigma}\ \Phi \in \Delta_n^G$ then we know that either $\boxed{\Sigma}\ \Phi \rightarrow \boxed{\Sigma'}\ \Phi'$ or $\Phi \equiv \Pi\, \epsilon$, as required.

Consider the case where there is an applicable update. We can choose the first update such that its predicate is applicable, and let i be this update's index. By Lemma 5.4 we know that $\mathtt{upd}(\Phi, u, b) \equiv \Phi^{\restriction}$, and hence that $\boxed{\Sigma_\emptyset}\ \mathtt{upd}(\Phi, u, b) \in \Delta_0^{G_2}$. We can use the inductive hypothesis to show that $\mathcal{G}^{k-1}(\Sigma_\emptyset, \mathtt{upd}(\Phi, u, b), \emptyset, \emptyset)$, as required. □

5.1.3 Summary

We can update programs whose effects are projections of Global Session Types; such situations correspond to a degree of global synchronisation. As such, at each point during the reduction we must consider whether the whole program can be so abstracted. At first glance this might seem a heavy requirement, however in practice we believe that we could use a check in/check out approach similar to that used in (Neamtiu and Hicks, 2009) to mitigate this cost. As an example we present an annotation of the program

$$
\begin{gathered}
t_p = \mu \underline{X}.\mathtt{dmod}_{\emptyset}(f(\mathtt{snd}(c_1, v_1).\underline{X})) \\
t_b = \mu \underline{X}'.\mathtt{dmod}_{\emptyset}(f'(\mathtt{rcv}(c_1)(x : T_1).\mathtt{snd}(c_2, v_2).\underline{X}')) \\
t_{co} = \mu \underline{X}''.\mathtt{dmod}_{\emptyset}(f''(\mathtt{rcv}(c_2)(x : T_2).\underline{X}''))
\end{gathered}
$$

Figure 5.5: Buffer System

$$
\begin{array}{ll}
1 & [\sigma_{\emptyset}]\ (t_p \parallel t_b \parallel t_{co}) \\
2 & \quad \rightarrow [c_1 \mapsto v_1]\ (\mathtt{dmod}_f(t_p) \parallel t_b \parallel t_{co}) \\
3 & \quad \rightarrow [\sigma_{\emptyset}]\ (\mathtt{dmod}_f(t_p) \parallel \mathtt{dmod}_{f'}(\mathtt{snd}(c_2, v_2).t_b) \parallel t_{co}) \\
4 & \quad \rightarrow [\sigma_{\emptyset}]\ (t_p \parallel \mathtt{dmod}_{f'}(\mathtt{snd}(c_2, v_2).t_b) \parallel t_{co}) \\
5 & \quad \rightarrow [\sigma_{\emptyset}]\ (t_p \parallel \mathtt{snd}(c_2, v_2).t_b \parallel t_{co}) \\
6 & \quad \rightarrow [c_1 \mapsto v_1]\ (\mathtt{dmod}_f(t_p) \parallel \mathtt{snd}(c_2, v_2).t_b \parallel t_{co}) \\
7 & \quad \rightarrow [c_1 \mapsto v_1, c_1 \mapsto v_2]\ (\mathtt{dmod}_f(t_p) \parallel t_b \parallel t_{co}) \\
8 & \quad \rightarrow [c_1 \mapsto v_1]\ (\mathtt{dmod}_f(t_p) \parallel t_b \parallel \mathtt{dmod}_{f''}(t_{co})) \\
9 & \quad \rightarrow \ldots
\end{array}
$$

Figure 5.6: Infinite Delay Using Round Robin Fair Scheduling

defined in Figure 4.4.

$$
\begin{gathered}
t_p = {}^1\mu \underline{X}.{}^2\mathtt{snd}(c_1, v_1).{}^3\mathtt{rcv}(c_2)(x : \textsc{Unit}).\underline{X} \\
t_{co} = {}^1\mu \underline{X}'.{}^2\mathtt{rcv}(c_1)(x : T_1).{}^3\mathtt{snd}(c_2, ()).\underline{X}'
\end{gathered}
\tag{5.3}
$$

We can then represent the code point $t_p \parallel t_{co}$ as 11, the code point:

$$
{}^2\mathtt{snd}(c_1, v_1).{}^3\mathtt{rcv}(c_2)(x : \textsc{Unit}).t_p \parallel t_{co} \tag{5.4}
$$

as 21, etc. We can then easily show that the effect of a program is the projection of a Global Session Type when a program is at point 11, 22, etc.

5.2 Local Update

While the Global Typability approach is applicable to all long running event processing loops, it suffers from the disadvantage that in some settings the update can be delayed indefinitely. One example where we may reasonably expect the indefinite delay of an update's application, even in the presence of fair scheduling, is a producer/consumer example with a buffer (Figure 5.5). If we take a round robin scheduling of this process, we obtain the reduction sequence shown in Figure 5.6. Here, after the initial point, we never reach a state where the system is the projection of a global session type; either a value is in the channel queues, or the threads are out of sync with respect to their $\mathtt{dmod}()$ annotations.

$$\begin{aligned} t_{p1} &= \mu\underline{X}.\texttt{dmod}_{\emptyset}(f(\texttt{snd}(c_1, v_1).\underline{X})) \\ t_{p2} &= \mu\underline{X}'.\texttt{dmod}_{\emptyset}(f'(\texttt{snd}(c_2, v_2).\underline{X}')) \\ t_{co} &= \mu\underline{X}''.\texttt{dmod}_{\emptyset}(f''(\texttt{rcv}(c_1)(x : T_1).\texttt{rcv}(c_2)(x : T_2).\underline{X}'')) \end{aligned}$$

Figure 5.7: Two Producer/Consumer System Without Acknowledgement

$$\begin{array}{lll} 1 & [\sigma_{\emptyset}] & (t_{p1} \parallel t_{p2} \parallel t_{co}), \psi_{\emptyset} \\ 2 & \rightarrow [c_1 \mapsto v_1] & (\texttt{dmod}_f(t_{p1}) \parallel t_{p2} \parallel t_{co}), \psi_{\emptyset} \\ 3 & \rightarrow [c_1 \mapsto v_1] & (\texttt{dmod}_f(t_{p1}) \parallel t_{p2} \parallel t_{co}), \psi \\ 4 & \rightarrow [c_1 \mapsto v_1] & (\texttt{dmod}_f(t_{p1}) \parallel t'_{p2} \parallel t_{co}), \psi' \\ 5 & \rightarrow [c_1 \mapsto v_1] & (t_{p1} \parallel t'_{p2} \parallel t_{co}), \psi' \\ 6 & \rightarrow [c_1 \mapsto v_1] & (t'_{p1} \parallel t'_{p2} \parallel t_{co}), \psi'' \end{array}$$

Figure 5.8: Introducing Deadlock Using Local Update

We should be able to update the system in Figure 5.5; intuitively we can see that if we update the producer first, then the buffer, then finally the consumer, that we should avoid errors and deadlock. This, however, requires updating the threads separately, and risks the old protocol interacting with the new.

Communication errors occur when a value is received on a channel that is not of the expected type. If we assume that the new program uses different channels to those in the old program, then neither can receive a value sent by the other, and we rule out communication errors completely. The problem then consists of guaranteeing liveness.

In this example, the producer never receives any values, and hence cannot be blocked (and hence deadlocked) while it is performing the old protocol. If we are to update the threads separately, this therefore seems like a good candidate to be the first updated. Once updated it may block waiting for values sent by other threads under the new protocol; this is not problematic if we can show that all threads will be updated to the new protocol, and that once they all are that the system behaves as a static message passing programs as in Chapter 3. We then have to guarantee that each thread in turn migrates to the new protocol.

The key restriction that we introduce in order to update a program using Local Update is that only one process can start the protocol. Consider the system in Figure 5.7. This program is a producer/consumer protocol with two senders and one receiver. Hence either the first or the second producer can evaluate first, and start the protocol. Consider an update that consists of three separate predicated updates. The predicates for t_{pi} are true when the thread reaches the top of its loop. The predicate for t_{co} is true when it reaches the top of its loop, its channel queues are empty, and the producer threads have already performed their updates. Then consider one reduction of this system

$$\begin{array}{ll}
1 & [\sigma_\emptyset]\ (t_p \parallel t_b \parallel t_{co})), \psi_\emptyset \\
2 & \quad \to \ldots \\
3 & \quad \to [c_1 \mapsto v_1]\ (t_p \parallel t_b \parallel t_{co}), \psi_\emptyset \\
4 & \quad \to [c_1 \mapsto v_1]\ (t_p \parallel t_b \parallel t_{co}), \psi_1, \psi_2, \psi_3 \\
5 & \quad \to [c_1 \mapsto v_1]\ (t'_p \parallel t_b \parallel t_{co}), \psi_2, \psi_3 \\
6 & \quad \to [\sigma_\emptyset]\ (t'_p \parallel \texttt{dmod}_{f_2}(c_2!\langle v_2\rangle;\ t_b) \parallel t_{co}), \psi_2, \psi_3 \\
7 & \quad \to [\sigma_\emptyset]\ (t'_p \parallel c_2!\langle v_2\rangle;\ t_b \parallel t_{co}), \psi_2, \psi_3 \\
8 & \quad \to [c_2 \mapsto v_2]\ (t'_p \parallel t_b \parallel t_{co}), \psi_2, \psi_3 \\
9 & \quad \to [c_2 \mapsto v_2]\ (t'_p \parallel t'_b \parallel t_{co}), \psi_3 \\
10 & \quad \to [\sigma_\emptyset]\ (t'_p \parallel t'_b \parallel \texttt{dmod}_{f_3}(t_{co})), \psi_3 \\
11 & \quad \to [\sigma_\emptyset]\ (t'_p \parallel t'_b \parallel t_{co}), \psi_3 \\
12 & \quad \to [\sigma_\emptyset]\ (t'_p \parallel t'_b \parallel t'_{co}), \psi_\emptyset
\end{array}$$

where:

$$\psi_i = p_i \mapsto (f_i \mapsto t_i) \qquad p_1(\ [\Sigma]\ \Phi, \psi) \overset{\text{def}}{=} (\varphi_1 \equiv \mu\underline{X}.\texttt{dmod}_\emptyset(f_1(\varphi'_1)))$$

$$p_2(\ [\Sigma]\ \Phi, \psi) \overset{\text{def}}{=} (\varphi_2 \equiv \mu\underline{X}.\texttt{dmod}_\emptyset(f_2(\varphi'_2)) \wedge \Sigma(c_1) = \emptyset \wedge \psi_1 \notin u)$$

$$p_3(\ [\Sigma]\ \Phi, \psi) \overset{\text{def}}{=} (\varphi_3 \equiv \mu\underline{X}.\texttt{dmod}_\emptyset(f_3(\varphi'_3)) \wedge \Sigma(c_2) = \emptyset \wedge \psi_1 \notin u \wedge \psi_2 \notin u)$$

Figure 5.9: Update Success Using Local Update

(Figure 5.8). After the first producer sends a value, the update is introduced. The second consumer updates straight away. The first consumer then drops its old `dmod()` and updates (line `6`). As the first producer is now working under the new protocol, it will never send a value on c_1. As the second producer has already sent a value under the old protocol, however, the consumer cannot update, but also will never reduce as it will never receive on channel c_1, hence the consumer is deadlocked.

We describe an example where Local Update preserves safety and liveness. We abuse notation and define the update predicates over effects rather than terms. Consider an update to the system defined in Figure 5.5, which updates the producer whenever it is at the top of its main loop, which updates the buffer when it is at the top of its main loop, it has no messages in its buffers, and the producer has already been updated, and updates the consumer when it is at the top of its main loop, it has no messages in its buffers, and both the producer and the buffer have already been updated. We present one possible reduction sequence of this system in Figure 5.9. Note how the producer is updated immediately (line `3`), but the buffer must reduce and consumer its messages before it can update (line `9`). Note also how the consumer waits until the buffer has updated before it performs its update, despite the fact it is at the top of its main loop and has no messages in its channel queues. In this example the consumer could update before the buffer without incident, however in a similar system, where the consumer

$$\Delta^{\restriction G}_{\underline{D}} \overset{\text{def}}{=} \bigcup_{n<\omega} {}_n\Delta^{\restriction G}_{\underline{D}} \qquad {}_0\Delta^{\restriction G}_{\underline{D}} \overset{\text{def}}{=} \{\, [\Sigma_\emptyset]\ \Phi^{\restriction G}_{\underline{D}} \}$$

$$\begin{aligned}
{}_{n+1}\Delta^{\restriction G}_{\underline{D}} \overset{\text{def}}{=}\ & \{\, \delta([c \mapsto T; \Sigma^k], \varphi'_b) \mid \forall j \in J.\delta(\Sigma^j, \varphi^j_b) \in {}_n\Delta^{\restriction G}_{\underline{D}} \wedge k \in J \} \\
& \cup \{\, \delta(\Sigma, \varphi'_a, \varphi'_b) \mid \forall j \in J.\delta(\Sigma, \varphi^j_a, \varphi^j_b) \in {}_n\Delta^{\restriction G}_{\underline{D}} \wedge \Sigma(c) = \emptyset \wedge d_a, d_b \in \underline{D} \} \\
& \cup \{\, \delta(\Sigma, \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b)) \mid \delta(\Sigma, \varphi^k_b) \in {}_n\Delta^{\restriction G}_{\underline{D}} \wedge \\
& \qquad G \restriction d_b \equiv \mathsf{dmod}_\emptyset(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b))))\} \\
& \cup \{\, \delta(\Sigma, \mathsf{dmod}_{\{f_{d_a}\}}(\varphi^k_a)) \mid \delta(\Sigma, \varphi^k_a) \in {}_n\Delta^{\restriction G}_{\underline{D}} \wedge \\
& \qquad G \restriction d_a \equiv \mathsf{dmod}_\emptyset(f_{d_a}(\oplus_K(c!(T_k);\ \varphi^k_a)))\} \\
& \cup\, {}_n\Delta^{\restriction G}_{\underline{D}}
\end{aligned}$$

where $c = d_a d_b$, $\emptyset \subset K \subseteq J$, $D = \overline{D} \cup \underline{D}$, and $\overline{D} \cap \underline{D} = \emptyset$, and

$$\varphi'_a = \begin{cases} \mathsf{dmod}_\emptyset(f_{d_a}(\oplus_K(c!(T_k);\ \varphi^k_a))) & \text{if } G \restriction d_a \equiv \mathsf{dmod}_\emptyset(f_{d_a}(\oplus_K(c!(T_k);\ \varphi^k_a))) \\ \oplus_K(c!(T_k);\ \varphi^k_a & \text{otherwise} \end{cases}$$

$$\varphi'_b = \begin{cases} \mathsf{dmod}_\emptyset(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b)))) & \text{if } G \restriction d_b \equiv \mathsf{dmod}_\emptyset(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b)))) \\ \&_J(c?(T_j);\ \varphi^j_b) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\Delta^{\restriction G}_{\overline{D}} \overset{\text{def}}{=}\ & \{\, [\Sigma]\ \textstyle\prod_{\overline{D}} \phi_{d_i} \mid [\Sigma]\ \prod_D \in \Delta^{\restriction G} \wedge \\
& \qquad \forall d \in \underline{D}.(\phi_d \equiv G \restriction d \wedge \forall c \in \mathsf{S}(\phi_d).\Sigma(c) = \emptyset)\}
\end{aligned}$$

Figure 5.10: Relaxed Valid Points Sets for Local Update

sends an acknowledgement to the buffer after receiving a value, if the consumer updates before the buffer then the buffer could be left part way through its execution, waiting for the acknowledgement, introducing deadlock.

Before we define Local Update we need to make some simplifying assumptions. We again will consider looping global types G^o that are of the form $\mu\underline{X}.f(G_o)$ where G_0 does not contain any instances of μ nor any region annotations, as in the previous section. We assume that the old program is one that can evolve to the projection of a global type G_1. We assume that the new program is a projection of the global type G_2, that no channels are shared between G_1 and G_2, and that G_1 and G_2 share the same set of roles. We assume that G_1 and G_2 are both of form G_o We define a partial order $<_{G_1}$ over participants d of a protocol G_1 denoting the order under which the participants can begin execution of a protocol G_1 (Figure 5.11). For example, for the Producer/Buffer/Consumer example we would define $d_p < d_b < d_{co}$. We prove that the relation defined in Figure 5.11 is a strict partial order in Lemma D.1.

$$\frac{d_1 <_G d_2}{d_1 <_{\mu\underline{X}.G} d_3} \qquad \frac{d_1 <_G d_2}{d_1 <_{f(G)} d_3} \qquad \frac{d_1 <_G d_2}{d_1 <_{\mathtt{dmod}_F(G)} d_3} \qquad \frac{d_1 <_G d_2 \quad d_2 <_G d_3}{d_1 <_G d_3}$$

$$\frac{d_1 <_{G_k} d_2 \quad d_2 \neq d, d'}{d_1 <_{d \to d' : c\langle \widetilde{T \mapsto G} \rangle} d_2} \qquad \frac{}{d_1 <_{d_1 \to d_2 : c\langle \widetilde{T \mapsto G_k} \rangle} d_2}$$

Figure 5.11: Definition of Participant Partial Order

We require that the order has a unique minimum element. In particular this will rule out examples such as:

$$\begin{aligned} \mu\underline{X}.\mathtt{dmod}(f(\quad & d_1 \to d_3 \colon c_1\langle T_1 \rangle; \\ & d_2 \to d_3 \colon c_2\langle T_2 \rangle; \\ & \mathsf{0} \\ )) & \end{aligned} \tag{5.5}$$

which represents the system in Figure 5.6. We omit the subscript when the meaning is clear. We only define $<_{G_1}$ for the Global Session Type G_1 of the original program. This order does not change as a given $\boxed{[\Sigma]}\ \Phi$ reduces. We assume that we can partition the roles in G_1 (and hence G_2) into two disjoint sets, where $\underline{D}$ denotes the roles of processes that have not yet been updated, and $\overline{D}$ denote the roles of processes that have been updated. We write $\underline{D} < \overline{D}$ when $\forall d \in \underline{D}, d' \in \overline{D}$ we have that $d \not< d'$.

We refer to the set of channels that are received upon in an effect ϕ as $\mathsf{R}(\phi)$, and refer to the set of channels that are sent upon by an effect ϕ as $\mathsf{S}(\phi)$. Define the Local Update predicate $(LU)\phi$ on code updates as:

$$\mathcal{LU}(\psi) \text{ iff } (\psi = (p_i(z_\Sigma, z_\Phi, z_\psi) \mapsto \widetilde{(f_{d_i} \mapsto t_{d_i})_D} \text{ and } \forall d_i \in D \,.\, \vdash t_{d_i} : G \restriction d_i) \tag{5.6}$$

where $p_i(z_\Sigma, z_\Phi, z_\psi)$ is $\forall c \in \mathsf{R}(z_{\phi_i}).z_\Sigma(c) = \emptyset \wedge \forall d_j <_{G_1} d_i.z_{\psi_j}$ is empty, ψ_j is the update for process projected from d_j, and D is the set of all roles used in G_1.

We relax the definition of the set of abstract configurations reducing towards G. We define the relaxed projection $\Phi_{\underline{D}}^{\restriction G}$ to mean that $\Phi \equiv \prod_D G \restriction G \restriction d_i$ for all participants d_i in a given D. Note that D need not comprise all roles in the given G.

We define the relaxed set of configurations for the non-updated roles $\underline{D}$ as $\Delta_{\underline{D}}^{\restriction G_1}$, where G_1 is the protocol of the old program (Figure 5.10). This definition permits the prefix of messages from processes whose role is not present in $\underline{D}$ when prefixing a receive action to a process that is present, along with prefixing send and receive actions to processes whose roles are both in $\underline{D}$. We perform the choice between possible branches, and the correct annotation of regions as previously.

We define the relaxed set of configurations for the updated roles $\overline{D}$ as $\Delta_{\overline{D}}^{\restriction G_2}$, where G_2 is the protocol of the new program (Figure 5.10). This set filters the non-relaxed definition

$\Delta^{\restriction G_2}$ (Figure 5.3) for those configurations where the effects of all the roles that are not in $\overline{D}$ are projections of the protocol G_2. That is, any roles that have not yet been updated should start at the beginning of their runs of G_2.

5.2.1 Properties and Proofs

In order to prove that Local Update implies Global Compatibility we make use of some technical lemmas.

Lemma 5.7. *Old Program Subject Reduction*

$$\forall \Sigma, \Phi, G, n, \underline{D}, \Sigma', \Phi'.\ [\Sigma]\ \Phi \in {}_n\Delta^{\restriction G}_{\underline{D}} \wedge [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow [\Sigma']\ \Phi' \in \Delta^{\restriction G}_{\underline{D}}$$

where the participants of G are D, and $\underline{D} <_G \overline{D}$.

Proof. This proof is similar that of Lemma 5.1, and proceeds by induction over n. The key differences are in the case when $n > 0$, in the third and fourth productions of ${}_{n+1}\Delta^{\restriction G}_{\underline{D}}$, as in Figure 5.10. Let $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b))$, where $\delta(\Sigma, \varphi^k_b) \in {}_n\Delta^{\restriction G}_{\underline{D}}$ and $G \restriction d_b \equiv \mathsf{dmod}_{\emptyset}(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b))))\}$. If the process that is reduced is $\mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b)$ then, by the reduction rules, we know that $[\sigma]\ \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b) \to [\Sigma]\ \varphi^k_b$. By the hypotheses we know that $\delta(\Sigma, \varphi^k_b) \in {}_n\Delta^{\restriction G}_{\underline{D}}$, as required. If another process is reduced than we proceed as in Lemma 5.1. The fourth case is similar to the third.

□

Lemma 5.8. *New Program Subject Reduction*

$$\forall \Sigma, \Phi, n, G, \overline{D}, \Sigma', \Phi'.\ [\Sigma]\ \Phi \in {}_n\Delta^{\restriction G}_{\overline{D}} \wedge [\Sigma]\ \Phi \to [\Sigma']\ \Phi' \Rightarrow [\Sigma']\ \Phi' \in \Delta^{\restriction G}_{\overline{D}}$$

where the participants of G are D, and $\underline{D} <_G \overline{D}$.

Proof. This proof directly follows that of Lemma 5.1, and proceeds by induction over n. Note that the key differences between the definition of ${}_n\Delta^{\restriction G}_{\overline{D}}$ and $\Delta^{\restriction G}$ (as in Lemma 5.1) are the restriction on which processes are included, and the fact that channels for non-included roles must have empty queues. As reduction does not add or remove processes, and cannot add messages to channels of processes that are not included (by principle channel allocation), then we can use essentially the same proof technique. □

We prove the straightforward property that program points in the valid set do not contain errors.

Lemma 5.9. *Local Update Safety*

$$\forall \Sigma, \Phi, G, \underline{D}, \overline{D}.\ [\Sigma]\ \Phi \in {}_n\Delta^{\restriction G}_{\underline{D}} \Rightarrow \mathtt{error} \notin \Phi$$

$$\forall \Sigma, \Phi, G, \underline{D}, \overline{D}.\ [\Sigma]\ \Phi \in {}_n\Delta^{\restriction G}_{\overline{D}} \Rightarrow \mathtt{error} \notin \Phi$$

where the participants of G *are* D*, and* $\underline{D} <_G \overline{D}$*.*

Proof. Assume the hypotheses. We then prove the conclusions by straightforward induction over n. This proof is similar to that of Lemma 3.10. □

We prove that each program point in the old program points set is either live, or has are relaxed projections of a global type:

Lemma 5.10. *Old Program Liveness*

$\forall \Sigma, \Phi, n, G, \underline{D}$ *If* $[\Sigma]\ \Phi \in {}_n\Delta^{\restriction G}_{\underline{D}}$ *then either* $\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \to [\Sigma']\ \Phi'$ *or* $[\Sigma]\ \Phi = [\Sigma_\emptyset]\ \Phi^{\restriction G}_{\underline{D}}$*, where the participants of* G *are* D*, and* $\underline{D} <_G \overline{D}$*.*

Proof. Assume the hypothesis. We then prove that one of the two conclusions hold, by induction over n.

Consider the case where $n = 0$. We straightforwardly have, by the definition of ${}_0\Delta^{\restriction G}_{\underline{D}}$, that $[\Sigma]\ \Phi = [\Sigma_\emptyset]\ \Phi^{\restriction G_1}_{\underline{D}}$.

Consider the case where $n > 0$. We consider each of the cases of the definition of ${}_n\Delta^{\restriction G}_{\underline{D}}$ in Figure 5.10. Consider the case where $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \varphi'_b)\}$. We know that $[c \mapsto T_k]\ \&_J(c?(T_j);\ \varphi^j_b) \to [\Sigma]\ \varphi^k_b$, and hence $\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \to [\Sigma']\ \Phi'$, as required. Consider the case where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \varphi'_a, \varphi'_b)\}$. By the reduction rules we know that $[\Sigma]\ \oplus_K\ (c!(T_k);\ \varphi^k_a \to [\Sigma; c \to T_k]\ \varphi^k_a$, and hence $\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \to [\Sigma']\ \Phi'$, as required. Consider the case where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b))\}$. By the definition of well formed global session types we know that the only region annotation that each process (with role d_i) will have is f_{d_i}. Hence by the reduction rules for dmod we can show that $\Sigma, \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b) \to \Sigma, \varphi^k_b$, and hence $\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \to [\Sigma']\ \Phi'$, as required. Consider the case where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_{\{f_{d_a}\}}(\varphi^k_a))\}$. By the definition of well formed global session types we know that the only region annotation that each process (with role d_i) will have is f_{d_i}. Hence by the reduction rules for dmod we can show that $\Sigma, \mathsf{dmod}_{\{f_{d_a}\}}(\varphi^k_a) \to \Sigma, \varphi^k_a$, and hence $\exists \Sigma', \Phi'.\ [\Sigma]\ \Phi \to [\Sigma']\ \Phi'$, as required. The case where $[\Sigma]\ \Phi \in {}_{n-1}\Delta^{\restriction G}_{\underline{D}}$ proceeds immediately from the inductive hypothesis. □

We need to make sure that updating a process of role d_i with an update $\mathcal{LU}(\psi)$ results in a process that conforms to the new protocol G_2.

Lemma 5.11. *$\forall G_1, d_i, T, \psi$. if $G_1 \restriction d_i \wr \emptyset \vdash t_{d_i} : T$ and $\mathcal{LU}(\psi)$ then $\exists G_2 . G_2 \restriction d_i \wr \emptyset \vdash$ $\mathsf{upd}(t_i, f_{d_i} \mapsto t'_{d_i}, \texttt{false}) : T$.*

Where $\psi = \widetilde{(p_i(z_\Sigma, z_\Phi, z_\psi)} \mapsto (f_{d_i} \mapsto t_{d_i})_D$, as $\mathcal{LU}(\psi)$ holds.

Proof. This proof follows that of Lemma 5.4. $\square$

We need to show that if we extend a relaxed configuration of updated processes with a newly updated process then the new configuration will be in relaxed set, extended by the new role:

Lemma 5.12. *$\forall \Sigma, \Phi, G_2, \overline{D}, d', \overline{D'}$. if $[\Sigma]\ \Phi \in \Delta^{\restriction G_2}_{\overline{D}}$ and $\overline{D'} = \overline{D \cup \{d'\}}$ then $[\Sigma]\ (\Phi || G_2 \restriction d') \in \Delta^{\restriction G_2}_{\overline{D'}}$.*

Proof. By the definition of $\Delta^{\restriction G_2}_{\overline{D}}$ we know that $[\Sigma]\ \Phi = [\Sigma]\ \prod_{\overline{D}} \phi_{d_i}$, where $[\Sigma]\ \prod_D \in \Delta^{\restriction G}$ and $\forall d \in \underline{D}.(\phi_d \equiv G \restriction d \wedge \forall c \in \mathsf{S}(\phi_d).\Sigma(c) = \emptyset)\}$. Hence we can straightforwardly show that $[\Sigma]\ (\Phi || G_2 \restriction d') \in \Delta^{\restriction G_2}_{\overline{D'}}$. $\square$

We also need to show that if we redact a relaxed configuration of non-updated processes by a newly updated process then the new configuration will be in the relaxed set, redacted by the updated role:

Lemma 5.13. *$\forall \Sigma, \Phi, G_1, \underline{D}, n, d', \underline{D'}$. if $[\Sigma]\ (\Phi || G_1 \restriction d') \in {}_n\Delta^{\restriction G_1}_{\underline{D'}}$. and $\forall c \in R(G \restriction d').\Sigma(c) = \emptyset$ and $\underline{D'} = \underline{D \cup \{d'\}}$ and $\nexists d \in D$ such that $d < d'$ then $[\Sigma]\ \Phi \in {}_n\Delta^{\restriction G_1}_{\underline{D}}$.*

Proof. Assume the hypotheses. We then show the conclusion by induction over the n in ${}_n\Delta^{\restriction G_1}_{\underline{D}}$. The case where $n = 0$ is immediate by the definition of ${}_0\Delta^{\restriction G_1}_{\underline{D}}$. We consider the case when $n > 0$. We then proceed by case analysis over the definition of ${}_n\Delta^{\restriction G_1}_{\underline{D'}}$, as in Figure 5.10.

Consider the case where $[\Sigma]\ \Phi \in \{\ \delta([c \mapsto T; \Sigma^k], \varphi'_b)\}$. We can then consider the identity of d'. If $d' = d_b$ then we have a contradiction, as the hypotheses state that $(c \mapsto T_k; \Sigma)(c) = \emptyset$, which is clearly not true. If $d' = d_b$ then by the inductive hypothesis we know that as $\delta(\Sigma, \phi_{d_b}) = [\Sigma]\ \ldots || \phi_b || \ldots || \phi_{d'} || \ldots \in {}_{n-1}\Delta^{\restriction G_1}_{\underline{D'}}$ then we know that $[\Sigma]\ \ldots || \phi_b || \ldots \in {}_{n-1}\Delta^{\restriction G_1}_{\underline{D}}$, and hence by the first construction of Figure 5.10 that $[\Sigma]\ \ldots || \phi'_b || \ldots \in {}_{n-1}\Delta^{\restriction G_1}_{\underline{D}}$, as required.

Consider the case where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \varphi'_a, \varphi'_b)\}$. We can then consider the identity of d'. If $d' = d_b$ and $\phi'_b = \&_J(c?(T_j);\ \varphi^j_b)$ then we have a contradiction, as the hypothesis requires that $\phi_{d'} \equiv G_1 \restriction d' \equiv \mathsf{dmod}_\emptyset(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b))))$. If $d' = d_b$ and $\phi'_b = \mathsf{dmod}_\emptyset(f_{d_b}(\&_J(c?(T_j);\ \varphi^j_b))))$ then we either have that $d_a \in \underline{D'}$ or not. If $d_a \in \underline{D'}$ then we have a contradiction, as $d_a < d_b$, and the hypothesis states that $\nexists d \in D'$ such that $d < d'$. Consider the case where $d_a \notin \underline{D'}$. This is a contradiction, as the hypothesis states that $d_a, d_b \in \underline{D}$. If $d' \neq d_b$ and $d' \neq d_a$ then the proof proceeds as when $d' \neq d_b$ in the first case.

The cases where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_{\{f_{d_b}\}}(\varphi^k_b))\}$ and where $[\Sigma]\ \Phi \in \{\ \delta(\Sigma, \mathsf{dmod}_{\{f_{d_a}\}}(\varphi^k_a))\}$ proceed similarly to the first case. The case where $[\Sigma]\ \Phi \in\ {}_n\Delta^{\restriction G}_{\underline{D}}$ proceeds directly from the inductive hypothesis. □

Finally, we can then use these lemmas to prove that Local Update implies Global Compatibility:

Theorem 5.14. *Local Update Theorem*

For all $k, \Sigma, P, \psi, \omega$, if $\mathcal{LU}(\psi, \omega)$ and we can form the partitions $\Sigma = \Sigma_1, \Sigma_2$ and $D = \underline{D} \uplus \overline{D}$ such that $[\Sigma_1]\ \prod_{\underline{D}} \phi_d \in \Delta^{\restriction G}_{\underline{D}}$ and $[\Sigma_2]\ \prod_{\overline{D}} \phi_d \in \Delta^{\restriction G}_{\overline{D}}$ and $\overline{D} < \underline{D}$ then $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$.

Proof. Assume the hypotheses. We then proceed by induction over k in $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$. The base case is trivial. Consider the case where $k < 0$. We prove each of the conjuncts of $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$, as defined in Figure 4.18, separately. In order to show $\texttt{error} \notin \Phi$ we appeal to Lemma 5.9 that shows that there are no errors in either $\prod_{\underline{D}} \phi_d$ or $\prod_{\overline{D}} \phi_d$. In order to show that $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega, \emptyset)$ we appeal to the fact that $\emptyset$ is the unit in update composition, and we have that $\mathcal{LU}(\psi, \omega, \emptyset)$ and hence can use the inductive hypothesis to show that $\mathcal{G}^{k-1}(\Sigma, \Phi, \psi, \omega)$. In order to show the first conjunct we perform a case analysis on whether an update is applicable or not.

Consider the case where an update is applicable. We must show that all reductions lead to safe program states, and the program is either live or empty. Consider a given reduction. This reduction must either be in the old program ($d \in \underline{D}$) or in the new program ($d \in \overline{D}$). If the former ($[\Sigma_1]\ \prod_{\underline{D}} \phi_d \rightarrow [\Sigma_1']\ \prod_{\underline{D}} \phi_d'$) then by Lemma 5.8 we know that $[\Sigma_1']\ \prod_{\underline{D}} \phi_d' \in\ {}_n\Delta^{\restriction G}_{\underline{D}}$. We then have a decomposition of $[\Sigma']\ \Phi'$ such that $\Sigma' = \Sigma_1', \Sigma_2$ and $[\Sigma_1']\ \prod_{\underline{D}} \phi_d' \in \Delta^{\restriction G}_{\underline{D}}$ and $[\Sigma_2]\ \prod_{\overline{D}} \phi_d \in \Delta^{\restriction G}_{\overline{D}}$, and hence can use the inductive hypothesis to show that $\mathcal{G}^{k-1}(\Sigma', \Phi', \psi, \omega)$. In order to show that the program is live or empty we appeal to Lemma 5.10, which denotes that the old program partition is either live or one of its processes are updatable. As there are no processes that are eligible for update, then the old program partition is live, as required.

Consider the case where an update is applicable. Let this process's role be d. By the update predicates we know that $\phi_d \equiv G_1 \restriction d$. By Lemma 5.11 we know that $G_2 \restriction d_i \wr \emptyset \vdash \mathsf{upd}(t_i, f_{d_i} \mapsto t'_{d_i}, \texttt{false}) : T$. Let $\underline{D} = \underline{D'} \uplus \{d\}$ and $\overline{D'} = \overline{D} \uplus \{d\}$. Then, by Lemmas 5.12 and 5.13 we can show that given the new partitions $\underline{D'}$ and $\overline{D'}$ that we have that $[\Sigma_1]\ \prod_{\underline{D'}} \phi_d \in \Delta^{\restriction G}_{\underline{D'}}$ and $[\Sigma_2]\ \prod_{\overline{D'}} \phi_d' \in \Delta^{\restriction G}_{\overline{D'}}$, where for each $d' \neq d$ we have that $\phi'_{d'} = \phi_{d'}$, and $\phi'_d = \mathsf{upd}(\phi_d, f_d \mapsto t'_d, \texttt{false})$, and that $\overline{D'} < \underline{D'}$. Hence can use the inductive hypothesis to show that $\mathcal{G}^{k-1}(\Sigma, \mathsf{upd}(\Phi, f_{d_i} \mapsto t'_{d_i}, b), \psi \setminus f_{d_i} \mapsto t'_{d_i}, \emptyset)$, as required. □

5.2.2 Summary

In order to be able to update threads separately we restrict the possible programs that be updated. In particular we consider only programs where each iteration of the event processing loop has a clear leader. The resulting system permits separate updates of individual threads, without any global synchronisation. In particular, this simplifies the checking of when the relevant update predicates are fulfilled: a separate monitor can be maintained for each thread, which only activates when a thread is at the top of its event processing loop, and in that case only needs to examine its channel queues and whether or not certain other threads have been updated.

5.3 Conclusions

We consider two possible approaches to providing update to blocking message passing programs: Global Typability and Local Update.

The main advantages of Global Typability are that it can handle most message passing programs that can be abstracted using Global Session Typing, that it can update to any program which can be abstracted using Global Session Typing, and that the update semantics is relatively straightforward to understand. The main disadvantages of Global Typability are that there is no guarantee that an update will ever be applied, and the cost of checking the update predicates.

The main advantages of Local Update are that threads can be updated separately, reducing the cost of checking update predicates (as no global information is required), and that assuming fair scheduling that updates will not be infinitely delayed. The disadvantage is that we restrict the form of updatable programs to those which have a unique thread that starts each iteration of the event processing loop. The degree to which this restriction will real out desired programs, in actual system development, is a topic for further research.

Chapter 6

Conclusions and Future Work

In this thesis we present techniques to prove communication safety and liveness in message passing programs. We re-prove key results from the session typing literature, using a novel inductive technique. We extend existing results of liveness and communication safety to non-blocking message passing systems.

We extend the above techniques to accommodate message passing programs that can be dynamically updated. Our key contribution are techniques which can be used to update multi-threaded programs without introducing deadlock. In particular we show how to, in certain situations, update threads separately, without requiring global synchronisation. We also present our intuition behind Update Liveness, the property that an update will not be delayed indefinitely.

The discovery of points at which update can be performed on more that one thread is widely accepted to be very difficult. We can utilise insights illuminated by our analysis to provide design patterns that are amenable to update. This has the potential to widely extend the ability to update multi-threaded programs.

In this chapter we provide a summary of how the above was achieved (Section 6.1) and the possibilities for future work (Section 6.2).

6.1 Summary

We began by considering examples that displayed type safety, type errors, liveness, and deadlock, for blocking message passing programs. From this we defined a general technique for ensuring safety and liveness, that amounts to model checking. We reproved existing results in Session Types work for blocking message passing programs using an intuitive inductive technique. In the course of this proof we identified the aspects that relied on the synchronisation behaviour of the blocking semantics. Then, when we wanted to prove similar results for non-blocking message passing, we were able to straightforwardly define the synchronisation behaviour required.

We then considered type safety and liveness for Dynamic Software Update. We analysed examples that display type safety, type errors, liveness, and deadlock, for updatable message passing programs. We then extended our general technique for proving safety and liveness of static programs to encompass updatable programs. These updates could be introduced at any time and applied at an arbitrary point specified by a predicate over the code and the state. We then considered techniques for performing update to programs that can be abstracted using Global Session Types. We proved that updates using these techniques would not introduce communication errors or deadlock. We made use of the synchronisation behaviour inherent in blocking message passing to co-ordinate when updates occurred. In certain cases this permitted us to update mutually dependent threads separately.

6.2 Future Work

While Dynamic Software Update has been informally considered since the early 1980s, it has only begun to be strongly focused upon in the last 10 years. As such a young field of inquiry, there are a wide variety of paths currently open for future work.

At a high methodological level it is unclear how much complexity is acceptable for programmers using DSU frameworks. The semantics of when an update occurs, particularly in systems where individual functions or class definitions are transformed rather than the entire program, is complex. As such, it is difficult for a non-specialist to reason about how state transformation functions behave at runtime. When updates are made more transparent, however, to the degree that state transformation functions are no longer needed, updates that change function or class signatures must be ruled out (Gregersen and Jorgensen, 2009). Clearly, given studies that indicate that at least 10% of all updates require such flexibility, this is not an acceptable restriction. Further study is needed to determine the acceptable level of complexity for users of DSU. One possible line of research is to give DSU tools to real programmers, and to get them experiment with updates to real programs, and to write the relevant transformation functions. One could then compare the functions written by programmers to those written by DSU specialists, and gather feedback on what was straightforward, and what was complex.

One approach to decrease the complexity of programming using DSU is to perform whole program transformation, starting the update at specified update points in the code, and re-entering the new code at specified entry points. While this technique significantly reduces the conceptual burden, it relies on programs reaching specific update points. This restriction is more consequential in multi-threaded programs; in a naive implementation there is no guarantee that the threads will ever all be at an updatable point.

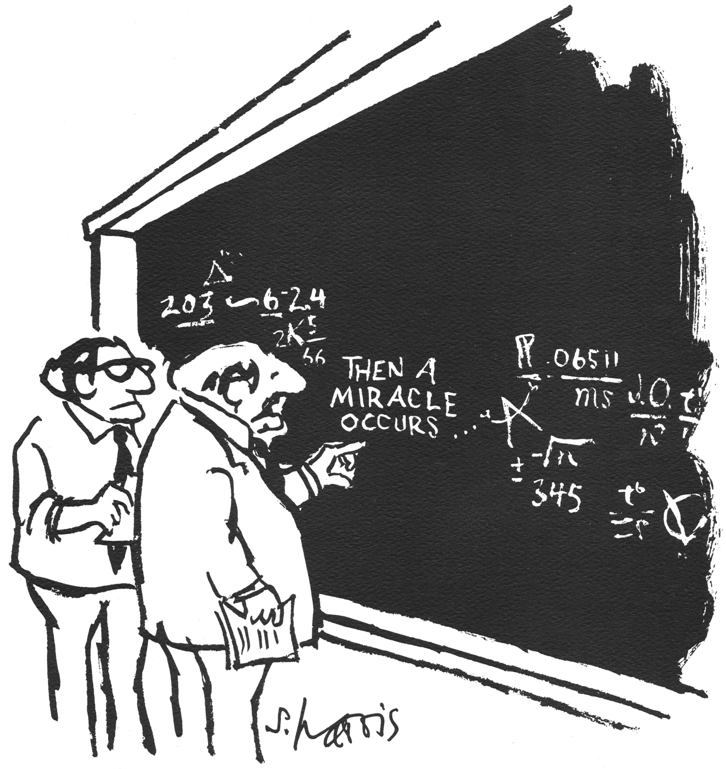
We could incorporate our analysis, which uses synchronisation behaviour, into work on whole program transformation in order to provide confidence that updates will not be delayed indefinitely.

Mobile computation devices, such as smart phones, run many programs (known as *apps*) concurrently. Unfortunately, due to hardware constraints, it is currently not possible to keep all of these apps in memory at the same time. As a result the OS will often shut down an app when the user wishes to run another. In order to maintain the illusion that an app has been in the background, rather than shut down and restarted, the app designers can make use of design pattern that allows them to save existing state to backing store, and to automatically load in that state when the app is restarted. This is a form of DSU (Hayden et al., 2011d), in particular one that uses a form of whole program transformation. As the mobile apps must run on limited hardware they are typically smaller than many production systems. As such it may be possible to perform case studies on how amenable they are to DSU on a large number of such apps, in order to obtain more general results.

Various behaviours have been identified when performing Dynamic Software Update that, while they are not necessarily errors in traditional safety properties, cause confusing or intuitively incorrect behaviour (Gregersen and Jorgensen, 2011). Some of these behaviours we may wish to consider erroneous, such as when classes are removed by a dynamic update, but live objects that belong to that class remain. Some of these behaviours may be erroneous or not, depending on the situation. Transient inconsistency occurs when an updated application is temporally in an unreachable runtime state; in Local Update, in the period between when the leading thread updates and the following threads do, the system is in a transiently inconsistent state, but will eventually reach a consistent state. Further study is required to determine where and when these phenomena are actually erroneous, particularly by examining the behaviour of real updated programs. Gregersen and Jorgensen also describe several design patterns that they posit will ameliorate such phenomena. Study into the impact of using such patterns on programmers, and on the efficacy of these patterns, is also required.

Session Typing techniques can be used to analyse the call graph of object oriented programs. As such, we should be able to straightforwardly extend our techniques to updating object oriented programs. There exists a significant body of work on updating object oriented programs, in particular (Subramanian, 2010) which utilises interrupts to detect safe update points and the garbage collector to provide object transformation. These techniques provide update facilities with minimal overhead. One major drawback, however, is that, in the interest of maintaining type safety in a straightforward manner, Subramanian only permits update to methods that are not active. This leads to some updates being delayed indefinitely. We believe that we could combine our more complex safety analysis with their efficient implementation to provide an update system that permits update to active methods, but still has minimal overhead. In addition, we would like to formalise our work on Update Liveness, and prove, for certain classes of Global Typability and Local Update, that updates are never delayed indefinitely.

Appendices



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Appendix A

General Formulation

In this chapter we provide the proofs of communication safety and liveness for the general formulation of dynamic programs. The proofs for the general formulation of static programs are instantiations of these proofs.

A.1 Environment Extension Lemma

Lemma A.1. *If $\varphi \wr \Gamma \vdash t \colon T$ then $\varphi \wr \Gamma, \Gamma' \vdash t \colon T$.*

Proof. By simultaneous induction over t and l. Note that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma') = \emptyset$

NB: as we type v using an empty environment it cannot be another variable x'.

Case: $t = n$

$$\frac{}{\epsilon \wr \Gamma \vdash n \colon \textsc{Int}}$$

We can trivially say that:

$$\frac{}{\epsilon \wr \Gamma, \Gamma' \vdash n \colon \textsc{Int}}$$

Case: $t = b$, $t = ()$, $t = r$, $t = l$

Similar to Case: $t = n$.

Case: $t = x$

$$\frac{}{\epsilon \wr \Gamma \vdash x \colon \Gamma(x)}$$

As $\Gamma(x)$ is defined we know that $x \in \mathsf{dom}(\Gamma)$. Since composition of type environments assumes disjoint domains then we can say that:

$$\frac{}{\epsilon \wr (\Gamma, \Gamma') \vdash x \colon (\Gamma, \Gamma')(x)}$$

Case: $t = X$

$$\frac{\Gamma(\underline{X}) = T_1 \xrightarrow{X} T_2}{\underline{X} \wr \Gamma \vdash \underline{X} \colon \Gamma(\underline{X})}$$

As $\Gamma(\underline{X})$ is defined we know that $\underline{X} \in \mathsf{dom}(\Gamma)$. Since composition of type environments assumes disjoint domains then we can say that:

$$\frac{(\Gamma, \Gamma')(\underline{X}) = T_1 \xrightarrow{X} T_2}{\underline{X} \wr (\Gamma, \Gamma') \vdash \underline{X} \colon (\Gamma, \Gamma')(\underline{X})}$$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \Gamma \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma \vdash l \colon T}$$

As this type rule places no restrictions on Γ then we can trivially show that:

$$\frac{\epsilon \wr \Gamma, \Gamma' \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma, \Gamma' \vdash l \colon T}$$

Case: l

$$\frac{\epsilon \wr \Gamma \vdash v_i \colon T_i}{\Gamma \vdash \alpha(\widetilde{v}) \colon \alpha(\widetilde{T})}$$

By induction we know that $\epsilon \wr \Gamma, \Gamma' \vdash v_i \colon T_i$. Hence we can say that:

$$\frac{\epsilon \wr \Gamma, \Gamma' \vdash v_i \colon T_i}{\Gamma, \Gamma' \vdash_{\psi} \alpha(\widetilde{v}) \colon \alpha(\widetilde{T})}$$

Case: $t = \texttt{rec}\, X(x : T).t$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash t \colon T_2 \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \texttt{rec}\, \underline{X}(x : T_1).t \colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

By induction we know that $\varphi \wr \Gamma, x\colon T_1, \underline{X}\colon T_1 \xrightarrow{\varphi} T_2, \Gamma' \vdash v_i\colon T_i$, where we α-convert x as necessary to ensure that they do not appear in Γ'. Hence we can say that:

$$\frac{\begin{array}{c}\varphi \wr \Gamma, \Gamma', x\colon T_1, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash t\colon T_2 \\ x \notin \mathsf{fv}(\Gamma, \Gamma'), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma, \Gamma' \vdash \mathtt{rec}\, \underline{X}(x : T_1).t\colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

Case: $t = t_1\, t_2$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma \vdash t_2\colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma \vdash t_1\, t_2\colon T_1}$$

By induction we know that:

$$\varphi_1 \wr \Gamma, \Gamma' \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma, \Gamma' \vdash t_2\colon T_2$$

Hence we can say that:

$$\frac{\varphi_1 \wr \Gamma, \Gamma' \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma, \Gamma' \vdash t_2\colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma, \Gamma' \vdash t_1\, t_2\colon T_1}$$

Case: $t = \mathtt{case}\, l\, \mathtt{in}\, \{\widetilde{T \mapsto t}\}$

$$\frac{\varphi_i \wr \Gamma \vdash t_i\colon T \qquad \epsilon \wr \Gamma \vdash v_i\colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \Gamma \vdash \mathtt{case}\, l\, \mathtt{in}\, \{\widetilde{T \mapsto t}\}\colon T}$$

By induction we know that

$$\varphi_i \wr \Gamma, \Gamma' \vdash t_i\colon T$$

Hence we can say that:

$$\frac{\varphi_i \wr \Gamma, \Gamma' \vdash t_i\colon T \qquad \epsilon \wr \Gamma, \Gamma' \vdash v_i\colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \Gamma, \Gamma' \vdash \mathtt{case}\, l\, \mathtt{in}\, \{\widetilde{T \mapsto t}\}\colon T}$$

Case: $t = \mathtt{error}$

This case can never be typed and hence can be disregarded as the hypothesis assumes that t is typed.

Case: $t = f(t)$

$$\frac{\varphi \wr \Gamma \vdash t\colon T}{f(\varphi) \wr \Gamma \vdash f(t)\colon T}$$

By induction we know that

$$\varphi \wr \Gamma, \Gamma' \vdash t \colon T$$

Hence we can say that:

$$\frac{\varphi \wr \Gamma, \Gamma' \vdash t \colon T}{f(\varphi) \wr \Gamma, \Gamma' \vdash f(t) \colon T}$$

Case: $t = \texttt{dmod}_{\widetilde{f}}(t)$

$$\frac{\varphi \wr \Gamma \vdash t \colon T}{\texttt{dmod}_{\widetilde{f}}(\varphi) \wr \Gamma \vdash \texttt{dmod}_{\widetilde{f}}(t) \colon T}$$

By induction we know that:

$$\varphi \wr \Gamma, \Gamma' \vdash t \colon T$$

Hence we can say that:

$$\frac{\varphi \wr \Gamma, \Gamma' \vdash t \colon T}{\texttt{dmod}_{\widetilde{f}}(\varphi) \wr \Gamma, \Gamma' \vdash \texttt{dmod}_{\widetilde{f}}(t) \colon T}$$

□

A.2 Substitution Lemma

Lemma A.2. *If*

$$\varphi \wr \Gamma, x\colon T' \vdash t\colon T \qquad \epsilon \wr \emptyset \vdash v\colon T'' \qquad T'[v/x] = T''$$

then

$$\varphi[v/x] \wr \Gamma[v/x] \vdash t[v/x]\colon T[v/x]$$

Proof. By induction over t

Case: $t = n$

$$\frac{}{\epsilon \wr \Gamma, x\colon T' \vdash n\colon \textsc{Int}}$$

we know that:

$$\epsilon[v/x] = \epsilon \qquad n[v/x] = n \qquad \textsc{Int}[v/x] = \textsc{Int} = T''$$

hence we can say:

$$\frac{}{\epsilon \wr \Gamma[v/x] \vdash n\colon \textsc{Int}}$$

Case: $t = b, t = (), t = l, t = X$

Similar to Case: $t = n$.

Case: $t = x'$

$$\frac{}{\epsilon \wr \Gamma' \vdash x'\colon \Gamma'(x')}$$

where $\Gamma' = \Gamma, x\colon T'$.

If $x' = x$:

$$\epsilon[v/x] = \epsilon \qquad x'[v/x] = v$$

We want to prove that $\epsilon \wr \Gamma[v/x] \vdash v\colon T'[v/x]$. We do this by a case split over the structure of T'

Case $T' = \textsc{Int}, \textsc{Bool}, \textsc{Unit}$

$T'[v/x] = T' = T''$

$\epsilon \wr \Gamma[v/x] \vdash v\colon T'$ using the TInt, TBool, or TUnit rule respectively

Case $T' = \mathsf{Res}\, r$

By the hypothesis we know that $\epsilon \wr \emptyset \vdash v \colon \mathsf{Res}\, r$.

As the type environment is empty of this type judgement is empty, the only way we could derive this fact is by using the TRes rule, and hence $v = r = r$.

The only T' such that $\mathsf{Res}\, r[v/x] = \mathsf{Res}\, v = T''$ are $\mathsf{Res}\, v$ and $\mathsf{Res}\, x$.

As here $T = T'$ we then know that T must be either $\mathsf{Res}\, v$ or $\mathsf{Res}\, x$.

We can easily show that:

$$\mathsf{Res}\, v[v/x] = \mathsf{Res}\, v \qquad \mathsf{Res}\, x[v/x] = \mathsf{Res}\, v$$

We then can use the TRes rule to show that:

$$\frac{}{\epsilon \wr \Gamma[v/x] \vdash v \colon \mathsf{Res}\, v}$$

Case $T' = T_1 \xrightarrow{\varphi} T_2$

By the hypothesis we know that:

$$\epsilon \wr \emptyset \vdash v \colon (T_1 \xrightarrow{\varphi} T_2)[v/x]$$

By the Environment Extension Lemma we can say:

$$\epsilon \wr \Gamma[v/x] \vdash v \colon (T_1 \xrightarrow{\varphi} T_2)[v/x]$$

as required.

If $x' \neq x$:

$$\frac{\epsilon[v/x] = \epsilon \qquad x'[v/x] = x' \qquad \Gamma(x')[v/x] = \Gamma[v/x](x')}{\epsilon \wr \Gamma[v/x] \vdash x' \colon \Gamma[v/x](x')}$$

Case: $t = r$

$$\frac{}{\epsilon \wr \Gamma, x \colon T' \vdash r \colon \mathsf{Res}\, r}$$

We know that $r \neq x$ and hence that

$$\epsilon[v/x] = \epsilon \qquad r[v/x] = r \qquad \mathsf{Res}\, r[v/x] = \mathsf{Res}\, r$$

We can then simply use the tRes rule:

$$\frac{}{\epsilon \wr \Gamma[v/x] \vdash r \colon \mathsf{Res}\, r}$$

Case: $t = \texttt{rec}\, X(x' : T_1).t'$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, x\colon T', x'\colon T_1, \underline{X}\colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t'\colon T_2 \\ x' \notin \mathsf{fv}(\Gamma, x\colon T'), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma, x\colon T' \vdash \texttt{rec}\, \underline{X}(x' : T_1).t'\colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

By induction, where $T'[v/x] = T''$:

$$\varphi[v/x] \wr (\Gamma, x'\colon T_1, \underline{X}\colon T_1 \xrightarrow{x'} T_2)[v/x] \vdash t'[v/x]\colon T_2[v/x]$$

As x' is not free in $t, (\Gamma, x\colon T')$ we can α convert it to ensure it does not clash with x, hence we have that:

$$\frac{\begin{array}{c}\varphi[v/x] \wr \Gamma[v/x], x''\colon T_1[v/x], \underline{X}\colon T_1[v/x] \xrightarrow{\underline{X}} T_2[v/x] \vdash t'[v/x]\colon T_2[v/x] \\ x'' \notin \mathsf{fv}(\Gamma[v/x]), \mathsf{fv}(T_1[v/x]), \mathsf{fv}(T_2[v/x]), \mathsf{fv}(\varphi[v/x]) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma[v/x] \vdash \texttt{rec}\, \underline{X}(x'' : T_1[v/x]).t'[v/x]\colon T_1[v/x] \xrightarrow{\mu \underline{X}.\varphi[v/x]} T_2[v/x]}$$

and hence:

$$\epsilon \wr \Gamma[v/x] \vdash (\texttt{rec}\, \underline{X}(x'' : T_1).t')[v/x]\colon (T_1 \xrightarrow{\mu x'.\varphi} T_2)[v/x]$$

Case: $t = t_1\, t_1$

$$\frac{\varphi_1 \wr \Gamma, x\colon T' \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma, x\colon T' \vdash t_2\colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma, x\colon T' \vdash t_1\, t_2\colon T_1}$$

By induction:

$$\varphi_1[v/x] \wr \Gamma[v/x] \vdash t_1[v/x]\colon T_1 \xrightarrow{\varphi_3} T_1[v/x]$$

$$\varphi_2[v/x] \wr \Gamma[v/x] \vdash t_2[v/x]\colon T_2[v/x]$$

Therefore we can say:

$$\frac{\varphi_1[v/x] \wr \Gamma[v/x] \vdash t_1[v/x]\colon T_2[v/x] \xrightarrow{\varphi_3[v/x]} T_1[v/x] \qquad \varphi_2[v/x] \wr \Gamma[v/x] \vdash t_2[v/x]\colon T_2[v/x]}{(\varphi_1[v/x];\, \varphi_2[v/x];\, \varphi_3[v/x]) \wr \Gamma[v/x] \vdash t_1[v/x]\, t_2[v/x]\colon T_1[v/x]}$$

and hence:

$$(\varphi_1;\, \varphi_2;\, \varphi_3)[v/x] \wr \Gamma[v/x] \vdash (t_1\, t_2)[v/x] \colon T_1[v/x]$$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \Gamma, x \colon T' \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma, x \colon T' \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By induction we have that $\epsilon \wr \Gamma[v/x] \vdash v_i[v/x] \colon T_i[v/x]$. Hence we can show, using the TAcc rule, that:

$$(\alpha(\widetilde{T}), T)[v/x] \wr \Gamma[v/x] \vdash (\alpha(\widetilde{T}), T)[v/x] \colon T[v/x]$$

Case: $t = \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3$

$$\frac{\begin{array}{c}\varphi_1 \wr \Gamma, x \colon T' \vdash t_1 \colon \textsc{Bool} \\ \varphi_2 \wr \Gamma, x \colon T' \vdash t_2 \colon T \quad \varphi_3 \wr \Gamma, x \colon T' \vdash t_3 \colon T\end{array}}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \Gamma, x \colon T' \vdash \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \colon T}$$

By induction:

$$\varphi_1[v/x] \wr \Gamma[v/x] \vdash t_1[v/x] \colon \textsc{Bool}[v/x]$$

$$\varphi_2[v/x] \wr \Gamma[v/x] \vdash t_2[v/x] \colon T[v/x] \qquad \varphi_3[v/x] \wr \Gamma[v/x] \vdash t_3[v/x] \colon T[v/x]$$

We can the use the TIf rule:

$$\frac{\begin{array}{c}\varphi_1[v/x] \wr \Gamma[v/x] \vdash t_1[v/x] \colon \textsc{Bool} \\ \varphi_2[v/x] \wr \Gamma[v/x] \vdash t_2[v/x] \colon T[v/x] \quad \varphi_3[v/x] \wr \Gamma[v/x] \vdash t_3[x/x] \colon T[v/x]\end{array}}{\varphi_1[v/x];\, (\varphi_2[v/x] \oplus \varphi_3[v/x]) \wr \Gamma[v/x] \vdash \texttt{if}\, t_1[v/x]\, \texttt{then}\, t_2[v/x]\, \texttt{else}\, t_3[x/x] \colon T[v/x]}$$

and hence:

$$\varphi_1;\, (\varphi_2 \oplus \varphi_3)[v/x] \wr \Gamma[v/x] \vdash (\texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3)[v/x] \colon T[v/x]$$

Case: $t = \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T_i \mapsto t_i}\}$

$$\frac{\varphi_i \wr \Gamma, x\colon T' \vdash t_i\colon T \qquad \epsilon \wr \Gamma, x\colon T' \vdash v_i\colon T_i \qquad T_i \neq \mathsf{Res}\,r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \Gamma, x\colon T' \vdash \texttt{case}\,l\,\texttt{in}\,\{\widetilde{T \mapsto t}\}\colon T}$$

By induction we know that:

$$\varphi_i[v/x] \wr \Gamma[v/x] \vdash t_i[v/x]\colon T[v/x]$$

We can then say, using the TCASE rule:

$$\frac{\varphi_i[v/x] \wr \Gamma[v/x] \vdash t_i[v/x]\colon T[v/x] \qquad T[v/\widetilde{x] \mapsto t}[v/x] = T_1[v/x] \mapsto t_1[v/x] \ldots T_n[v/x] \mapsto t_n[v/x]}{\&(\alpha(\widetilde{v})[v/x], T_i[v/x]);\ \varphi_i[v/x] \wr \Gamma[v/x] \vdash \texttt{case}\,\alpha(\widetilde{v})[v/x]\,\texttt{in}\,\{T[v/\widetilde{x] \mapsto t}[v/x]\}\colon T[v/x]} \text{(:TC}$$

which implies:

$$(\&(\alpha(\widetilde{v}), T_i);\ \varphi_i)[v/x] \wr \Gamma[v/x] \vdash \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}[v/x]\colon T[v/x]$$

Case: $t = \texttt{error}$

This case can never be typed and hence can be disregarded as the hypothesis assumes that t is typed.

Case: $t = \texttt{dmod}_{F_1}(t')$

$$\frac{\varphi \wr \Gamma, x\colon T' \vdash t'\colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi) \wr \Gamma, x\colon T' \vdash \texttt{dmod}_{\gamma_1}(t')\colon \textsc{Unit}}$$

By induction we know that:

$$\varphi[v/x] \wr \Gamma[v/x] \vdash t'[v/x]\colon \textsc{Unit}$$

Hence we can say that:

$$\frac{\varphi[v/x] \wr \Gamma[v/x] \vdash t'[v/x]\colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi[v/x]) \wr \Gamma[v/x] \vdash \texttt{dmod}_{\gamma_1}(t'[v/x])\colon \textsc{Unit}}$$

which implies:

$$\texttt{dmod}_{\gamma_1}(\varphi)[v/x] \wr \Gamma[v/x] \vdash \texttt{dmod}_{F_1}(t')[v/x]\colon \textsc{Unit}$$

Case: $t = f(t)$

Similar to case $t = \texttt{dmod}_{F_1}(t)$. $\square$

A.3 Recursive Variable Substitution Lemma

Lemma A.3. *If*

$$\varphi \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash t\colon T \qquad \epsilon \wr \emptyset \vdash v\colon T_1 \xrightarrow{\varphi'} T_2 \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)$$

then:

$$\varphi[\varphi'/\underline{X}] \wr \Gamma \vdash t[v/\underline{X}]\colon T[\varphi'/\underline{X}]$$

Proof. By induction over t

Case: $t = n$

$$\frac{}{\epsilon \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash n\colon \textsc{Int}}$$

we know that:

$$\epsilon[\varphi'/\underline{X}] = \epsilon \qquad n[v/\underline{X}] = n \qquad \textsc{Int}[\varphi'/\underline{X}] = \textsc{Int}$$

hence we can say:

$$\frac{}{\epsilon \wr \Gamma \vdash n\colon \textsc{Int}}$$

Case: $t = b, t = (), t = l, t = x, t = r$

Similar to Case: $t = n$.

Case: $t = X'$

$$\frac{\Gamma'(\underline{X'}) = T_1 \xrightarrow{X'} T_2}{\underline{X'} \wr \Gamma' \vdash \underline{X'}\colon \Gamma'(\underline{X'})}$$

where $\Gamma' = \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2$.

If $\underline{X'} = \underline{X}$:

$$\underline{X}[\varphi'/\underline{X}] = \varphi' \qquad \underline{X'}[v/\underline{X}] = v \qquad T_1 \xrightarrow{X} T_2[\varphi'/\underline{X}] = T_1[\varphi'/\underline{X}] \xrightarrow{\varphi'} T_2[\varphi'/\underline{X}] = T_1 \xrightarrow{\varphi'} T_2$$

as $\underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)$.

By the hypothesis we know that:

$$\epsilon \wr \emptyset \vdash v \colon T_1 \xrightarrow{\varphi'} T_2$$

and hence, using the Environment Extension Lemma, that:

$$\underline{X}[\epsilon/\underline{X}] \wr \Gamma \vdash v \colon T_1 \xrightarrow{\underline{X}} T_2[\varphi'/\underline{X}]$$

If $\underline{X}' \neq \underline{X}$:

$$\underline{X}'[\varphi'/\underline{X}] = \underline{X}' \qquad \underline{X}'[v/\underline{X}] = \underline{X}' \qquad T_1 \xrightarrow{\underline{X}'} T_2[\varphi/\underline{X}] = T_1 \xrightarrow{\underline{X}'} T_2$$

and hence that:

$$\underline{X}'[\varphi'/\underline{X}] \wr \Gamma \vdash \underline{X}' \colon T_1 \xrightarrow{\underline{X}'} T_2[\varphi'/\underline{X}]$$

Case: $t = \texttt{rec}\, X'(x' : T_1).t'$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2, x' \colon T_3, \underline{X}' \colon T_3 \xrightarrow{\underline{X}'} T_4 \vdash t' \colon T_4 \\ x' \notin \mathsf{fv}(\Gamma, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2), \mathsf{fv}(T_3), \mathsf{fv}(T_4), \mathsf{fv}(\varphi) \qquad \underline{X}' \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash \texttt{rec}\, \underline{X}'(x' : T_3).t' \colon T_3 \xrightarrow{\mu \underline{X}'.\varphi} T_4}$$

By induction, where $T'[v/x] = T''$:

$$\varphi[\varphi'/\underline{X}] \wr (\Gamma, x' \colon T_3, \underline{X} \colon T_3 \xrightarrow{\underline{X}'} T_4)[v/x] \vdash t'[v/\underline{X}] \colon T_4[\varphi'/\underline{X}]$$

As $\underline{X}'$ is not free in $t, (\Gamma, x \colon T')$ we can α convert it to ensure it does not clash with $\underline{X}$, hence we have that:

$$\frac{\begin{array}{c}\varphi[\varphi'/\underline{X}] \wr \Gamma, x' \colon T_1[v/\underline{X}], \underline{X} \colon T_1[v/\underline{X}] \xrightarrow{\underline{X}} T_2[\varphi'/\underline{X}] \vdash t'[v/\underline{X}] \colon T_2[\varphi'/\underline{X}] \\ x' \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1[v/\underline{X}]), \mathsf{fv}(T_2[\varphi'/\underline{X}]), \mathsf{fv}(\varphi[\varphi'/\underline{X}]) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \texttt{rec}\, \underline{X}(x' : T_1[v/\underline{X}]).t'[v/\underline{X}] \colon T_1[v/\underline{X}] \xrightarrow{\mu \underline{X}.\varphi[\varphi'/\underline{X}]} T_2[\varphi'/\underline{X}]}$$

and hence:

$$\epsilon[\varphi'/\underline{X}] \wr \Gamma \vdash (\texttt{rec}\, \underline{X}(x'' : T_1).t')[v/\underline{X}] \colon (T_1 \xrightarrow{\mu x'.\varphi} T_2)[\varphi'/\underline{X}]$$

Case: $t = t_1\, t_1$

$$\frac{\varphi_1 \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash t_2\colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash t_1\, t_2\colon T_1}$$

By induction:

$$\varphi_1[\varphi'/\underline{X}] \wr \Gamma \vdash t_1[v/\underline{X}]\colon T_1 \xrightarrow{\varphi_3} T_1[\varphi'/\underline{X}]$$

$$\varphi_2[\varphi'/\underline{X}] \wr \Gamma \vdash t_2[v/\underline{X}]\colon T_2[\varphi'/\underline{X}]$$

Therefore we can say:

$$\frac{\varphi_1[\varphi'/\underline{X}] \wr \Gamma \vdash t_1[v/\underline{X}]\colon T_2[\varphi'/\underline{X}] \xrightarrow{\varphi_3[\varphi'/\underline{X}]} T_1[\varphi'/\underline{X}] \qquad \varphi_2[\varphi'/\underline{X}] \wr \Gamma \vdash t_2[v/\underline{X}]\colon T_2[\varphi'/\underline{X}]}{(\varphi_1[\varphi'/\underline{X}];\, \varphi_2[\varphi'/\underline{X}];\, \varphi_3[\varphi'/\underline{X}]) \wr \Gamma \vdash t_1[v/\underline{X}]\, t_2[v/\underline{X}]\colon T_1[\varphi'/\underline{X}]}$$

and hence:

$$(\varphi_1;\, \varphi_2;\, \varphi_3)[\varphi'/\underline{X}] \wr \Gamma \vdash (t_1\, t_2)[v/\underline{X}]\colon T_1[\varphi'/\underline{X}]$$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash v_i\colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma, \underline{X}\colon T_1 \xrightarrow{X} T_2 \vdash (\alpha(\widetilde{v}), T)\colon T}$$

By induction we have that $\epsilon \wr \Gamma \vdash v_i[v/\underline{X}]\colon T_i[\varphi'/\underline{X}]$. Hence we can show, using the TAcc rule, that:

$$(\alpha(\widetilde{T}), T)[\varphi'/\underline{X}] \wr \Gamma \vdash (\alpha(\widetilde{T}), T)[v/\underline{X}]\colon T[\varphi'/\underline{X}]$$

Case: $t = \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3$

$$\frac{\begin{array}{c}\varphi_1 \wr \Gamma, x\colon T' \vdash t_1\colon \textsc{Bool}\\ \varphi_2 \wr \Gamma, x\colon T' \vdash t_2\colon T \quad \varphi_3 \wr \Gamma, x\colon T' \vdash t_3\colon T\end{array}}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \Gamma, x\colon T' \vdash \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3\colon T}$$

By induction:

$$\varphi_1[\varphi'/\underline{X}] \wr \Gamma \vdash t_1[v/\underline{X}]\colon \textsc{Bool}[\varphi'/\underline{X}]$$

$$\varphi_2[\varphi'/\underline{X}] \wr \Gamma \vdash t_2[v/\underline{X}]\colon T[\varphi'/\underline{X}] \qquad \varphi_3[\varphi'/\underline{X}] \wr \Gamma \vdash t_3[v/\underline{X}]\colon T[\varphi'/\underline{X}]$$

We can the use the TIF rule:

$$\frac{\begin{array}{c}\varphi_1[\varphi'/\underline{X}] \wr \Gamma \vdash t_1[v/\underline{X}] \colon \textsc{Bool} \\ \varphi_2[v/x] \wr \Gamma \vdash t_2[v/\underline{X}] \colon T[\varphi'/\underline{X}] \quad \varphi_3[v/x] \wr \Gamma \vdash t_3[x/\underline{X}] \colon T[\varphi'/\underline{X}]\end{array}}{\varphi_1[\varphi'/\underline{X}];\ (\varphi_2[v/x] \oplus \varphi_3[v/x]) \wr \Gamma \vdash \texttt{if}\, t_1[v/\underline{X}]\, \texttt{then}\, t_2[v/\underline{X}]\, \texttt{else}\, t_3[x/\underline{X}] \colon T[\varphi'/\underline{X}]}$$

and hence:

$$\varphi_1;\ (\varphi_2 \oplus \varphi_3)[\varphi'/\underline{X}] \wr \Gamma \vdash (\texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3)[v/\underline{X}] \colon T[\varphi'/\underline{X}]$$

Case: $t = \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T_i \mapsto t_i}\}$

$$\frac{\varphi_i \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash t_i \colon T \qquad \epsilon \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash \texttt{case}\, \alpha(\widetilde{T})\, \texttt{in}\, \{\widetilde{T \mapsto t}\} \colon T}$$

By induction we know that:

$$\varphi_i[\varphi'/\underline{X}] \wr \Gamma \vdash t_i[v/\underline{X}] \colon T[\varphi'/\underline{X}]$$

We can then say, using the TCASE rule:

$$\frac{\varphi_i[\varphi'/\underline{X}] \wr \Gamma \vdash t_i[v/\underline{X}] \colon T[\varphi'/\underline{X}] \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{v})[\varphi'/\underline{X}], T_i[\varphi'/\underline{X}]);\ \varphi_i[\varphi'/\underline{X}] \wr \Gamma \vdash \texttt{case}\, \alpha(\widetilde{v})[v/\underline{X}]\, \texttt{in}\, \{\widetilde{T[v/\underline{X}] \mapsto t[v/\underline{X}]}\} \colon T[\varphi'/\underline{X}]} \text{(:TCASE)}$$

which implies that:

$$(\&(\alpha(\widetilde{v}), T_i);\ \varphi_i)[\varphi'/\underline{X}] \wr \Gamma[v/x] \vdash \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T \mapsto t}\}[v/\underline{X}] \colon T[\varphi'/\underline{X}]$$

Case: $t = \texttt{error}$

This case can never be typed and hence can be disregarded as the hypothesis assumes that t is typed.

Case: $t = \mathtt{dmod}_{F_1}(t')$

$$\frac{\varphi \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash t' \colon \textsc{Unit}}{\mathtt{dmod}_{\gamma_1}(\varphi) \wr \Gamma, \underline{X} \colon T_1 \xrightarrow{X} T_2 \vdash \mathtt{dmod}_{\gamma_1}(t') \colon \textsc{Unit}}$$

By induction we know that:

$$\varphi[\varphi'/\underline{X}] \wr \Gamma \vdash t'[v/\underline{X}] \colon \textsc{Unit}[\varphi'/\underline{X}]$$

Hence we can say that:

$$\frac{\varphi[\varphi'/\underline{X}] \wr \Gamma \vdash t'[v/\underline{X}] \colon \textsc{Unit}[\varphi'/\underline{X}]}{\mathtt{dmod}_{\gamma_1}(\varphi[\varphi'/\underline{X}]) \wr \Gamma \vdash \mathtt{dmod}_{\gamma_1}(t'[v/\underline{X}]) \colon \textsc{Unit}[\varphi'/\underline{X}]}$$

which implies:

$$\mathtt{dmod}_{\gamma_1}(\varphi)[\varphi'/x] \wr \Gamma \vdash \mathtt{dmod}_{F_1}(t')[v/\underline{X}] \colon \textsc{Unit}$$

Case: $t = f(t)$

Similar to case $t = \mathtt{dmod}_{F_1}(t)$. □

A.4 Update Equivalence Lemma

Lemma A.4.

$$\varphi \equiv \varphi' \Rightarrow \mathsf{upd}(\varphi, u, b) \equiv \mathsf{upd}(\varphi', u, b)$$

Proof. By induction over $\equiv$

Case $\varphi \equiv \epsilon;\ \varphi$

$$\begin{aligned} \mathsf{upd}(\epsilon;\ \varphi, u, b) &= \epsilon;\ \mathsf{upd}(\varphi, u, b) \\ &\equiv \mathsf{upd}(\varphi, u, b) \end{aligned}$$

Case $\mu X.\varphi \equiv \varphi[\mu X.\varphi/X]$

$$\begin{aligned} \mathsf{upd}(\mu\underline{X}.\varphi, u, b) &= \mu\underline{X}.\mathsf{upd}(\varphi, u, b) \\ &\equiv \mathsf{upd}(\varphi, u, b)[\mu\underline{X}.\mathsf{upd}(\varphi, u, b)/\underline{X}] \end{aligned}$$

Case $\varphi = \varphi_1;\ \varphi_2$

By induction we know that:

$$\varphi_i' \equiv \varphi_i \Rightarrow \mathsf{upd}(\varphi_i, u, b) \equiv \mathsf{upd}(\varphi_i', u, b)$$

We know that:

$$\mathsf{upd}(\varphi_1;\ \varphi_2, u, b) = \mathsf{upd}(\varphi_1, u, b);\ \mathsf{upd}(\varphi_2, u, b)$$

Hence we have that:

$$\mathsf{upd}(\varphi_1;\ \varphi_2, u, b) \equiv \mathsf{upd}(\varphi_1';\ \varphi_2', u, b)$$

Case $\varphi = \varphi_1 \oplus \varphi_2, \varphi_1 \& \varphi_2, \mu X.\varphi_1, f(\varphi_1), \mathtt{dmod}_F(\varphi_1)$

Similar to case $\varphi = \varphi_1;\ \varphi_2$.

Case $\varphi = f(\varphi_1)$

Case $b = \texttt{true}$

By the definition of `upd` we know that:

$$\mathsf{upd}(f(\varphi_1), u, \texttt{false}) = f(\varphi')$$

where:

$$f \mapsto t' \notin u \qquad \varphi' \wr \emptyset \vdash t' :$$

Hence we have that:

$$\mathsf{upd}(f(\varphi_1), u, \texttt{true}) = \mathsf{upd}(f(\varphi_1'), u, \texttt{true}) = f(\varphi')$$

Case $b = \texttt{false}$

By induction we know that:

$$\varphi_1' \equiv \varphi_1 \Rightarrow \mathsf{upd}(\varphi_1, u, b) \equiv \mathsf{upd}(\varphi_1', u, b)$$

By the definition of `upd` we know that:

$$\mathsf{upd}(f(\varphi_1), u, \texttt{false}) = f(\mathsf{upd}(\varphi_1, u, b))$$

Hence we have that:

$$\mathsf{upd}(f(\varphi_1), u, \texttt{false}) \equiv \mathsf{upd}(f(\varphi_1'), u, \texttt{false})$$

Case $\varphi = \mathtt{dmod}_F(\varphi_1)$

Case $\gamma \cap \mathsf{dom}(u) = \emptyset$

By the definition of `upd` we know that:

$$\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1), u, b) = \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi_1, u, \mathtt{true}))$$

By induction we know that:

$$\varphi_1' \equiv \varphi_1 \Rightarrow \mathsf{upd}(\varphi_1, u, b) \equiv \mathsf{upd}(\varphi_1', u, b)$$

Hence we have that:

$$\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1), u, \mathtt{true}) \equiv \mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1'), u, \mathtt{true})$$

Case $\gamma \cap \mathsf{dom}(u) \neq \emptyset$

By the definition of `upd` we know that:

$$\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1), u, b) = \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi_1, u, \mathtt{false}))$$

By induction we know that:

$$\varphi_1' \equiv \varphi_1 \Rightarrow \mathsf{upd}(\varphi_1, u, b) \equiv \mathsf{upd}(\varphi_1', u, b)$$

Hence we have that:

$$\mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1), u, \mathtt{false}) \equiv \mathsf{upd}(\mathtt{dmod}_\gamma(\varphi_1'), u, \mathtt{false})$$

$\square$

A.5 Expression Update Consistency Lemma

Lemma A.5. *If:*

$$\varphi \wr \Gamma \vdash t \colon T$$

then:

$$\mathsf{upd}(\varphi, u, b) \wr \Gamma \vdash \mathsf{upd}(t, u, b) \colon T' \qquad T \equiv_{eff} T'$$

$$\frac{T_1 = T_2}{T_1 \equiv_{eff} T_2} \qquad \frac{T_1 \equiv_{eff} T_1' \qquad T_2 \equiv_{eff} T_2'}{T_1 \xrightarrow{\varphi} T_2 \equiv_{eff} T_1' \xrightarrow{\varphi'} T_2'}$$

Proof: by induction over t.

Case: $t = n$

$$\frac{}{\epsilon \wr \Gamma \vdash n \colon \textsc{Int}}$$

By the definition of upd:

$$\mathsf{upd}(n, u, b) \overset{\text{def}}{=} n \qquad \mathsf{upd}(\epsilon, u, b) \overset{\text{def}}{=} \epsilon$$

Hence we can say that:

$$\mathsf{upd}(\epsilon, u, b) \wr \Gamma \vdash \mathsf{upd}(n, u, b) \colon \textsc{Int}$$

Case: $t = b$, $t = ()$, $t = r$, $t = x$, $t = (\alpha(\widetilde{v}), T), t = X$

Similar to case $t = n$.

Case: $t = \texttt{rec}\, X(x : T).t_1$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t_1 \colon T_2 \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \texttt{rec}\, \underline{X}(x : T_1).t_1 \colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

By the definition of upd:

$$\mathsf{upd}(\texttt{rec}\, \underline{X}(x : T).t, u, b) \overset{\text{def}}{=} \texttt{rec}\, \underline{X}(x : T).\mathsf{upd}(t, u, b) \qquad \mathsf{upd}(\epsilon, u, b) \overset{\text{def}}{=} \epsilon$$

By induction we know that:

$$\mathsf{upd}(\varphi, u, b) \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\varphi} T_2 \vdash \mathsf{upd}(t_1, u, b) \colon T_2' \qquad T_2 \equiv_{eff} T_2'$$

Hence we can say that:

$$\frac{\begin{array}{c}\mathsf{upd}(\varphi,u,b) \wr \Gamma, x\colon T_1, \underline{X}\colon T_1 \xrightarrow{\underline{X}} T_2' \vdash \mathsf{upd}(t_1,u,b)\colon T_2' \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2'), \mathsf{fv}(\mathsf{upd}(\varphi,u,b)) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \mathtt{rec}\,\underline{X}(x : T_1).\mathsf{upd}(t_1,u,b)\colon T_1 \xrightarrow{\mu\underline{X}.\mathsf{upd}(\varphi,u,b)} T_2'}$$

which implies that:

$$\mathsf{upd}(\epsilon,u,b) \wr \Gamma \vdash \mathsf{upd}(\mathtt{rec}\,\underline{X}(x : T).t, u, b)\colon T_1 \xrightarrow{\mu\underline{X}.\mathsf{upd}(\varphi,u,b)} T_2'$$

where $T_1 \xrightarrow{\varphi} T_2 \equiv_{eff} T_1 \xrightarrow{\mu\underline{X}.\mathsf{upd}(\varphi,u,b)} T_2'$ as $T_1 = T_1$ and $T_2 \equiv_{eff} T_2'$ and

$$\frac{T_1 \equiv_{eff} T_1' \qquad T_2 \equiv_{eff} T_2'}{T_1 \xrightarrow{\varphi} T_2 \equiv_{eff} T_1' \xrightarrow{\varphi'} T_2'}$$

Case: $t = t_1\, t_2$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma \vdash t_2\colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma \vdash t_1\, t_2\colon T_1}$$

By the definition of upd:

$$\mathsf{upd}(t_1\, t_2, u, b) \overset{\text{def}}{=} \mathsf{upd}(t_1,u,b)\, \mathsf{upd}(t_2,u,b)$$

$$\mathsf{upd}(\varphi_1;\, \varphi_2;\, \varphi_3, u, b) \overset{\text{def}}{=} \mathsf{upd}(\varphi_1,u,b);\, \mathsf{upd}(\varphi_2,u,b);\, \mathsf{upd}(\varphi_3,u,b)$$

By induction we know that:

$$\mathsf{upd}(\varphi_1,u,b) \wr \Gamma \vdash \mathsf{upd}(t_1,u,b)\colon T_2' \xrightarrow{\mathsf{upd}(\varphi_3,u,b)} T_1'$$

$$\mathsf{upd}(\varphi_2,u,b) \wr \Gamma \vdash \mathsf{upd}(t_2,u,b)\colon T_2'$$

where $T_i \equiv_{eff} T_i'$. Hence we can say that:

$$\frac{\varphi_1' \wr \Gamma \vdash t_1'\colon T_2' \xrightarrow{\varphi_3'} T_1' \qquad \varphi_2' \wr \Gamma \vdash t_2'\colon T_2'}{(\varphi_1';\, \varphi_2';\, \varphi_3') \wr \Gamma \vdash t_1'\, t_2'\colon T_1'}$$

where $t_i' = \mathsf{upd}(t_i,u,b), \varphi_i' = \mathsf{upd}(\varphi_i,u,b), T_i \equiv_{eff} T_i'$.

Case: $t = \mathtt{if}\, t\, \mathtt{then}\, t\, \mathtt{else}\, t, \mathtt{dmod}_{F_1}(t)$

Similar to case $t = t_1\, t_2$

Case: $t = \mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{\widetilde{T \mapsto t'}\}$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\,r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \Gamma \vdash \mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{\widetilde{T \mapsto t}\} \colon T}$$

By the definition of upd:

$$\mathsf{upd}(\mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{\widetilde{T \mapsto t}\}, u, b) \overset{\text{def}}{=} \mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{T \mapsto \widetilde{\mathsf{upd}(t_1, u, b)}\}$$

$$\mathsf{upd}(\&\varphi_i, u, b) \overset{\text{def}}{=} \&\mathsf{upd}(\varphi_i, u, b)$$

By induction we have that:

$$\mathsf{upd}(\varphi_i, u, b) \wr \Gamma \vdash \mathsf{upd}(t_i, u, b) \colon T'$$

where $T \equiv_{eff} T'$. Hence we can say that:

$$\frac{\mathsf{upd}(\varphi_i, u, b) \wr \Gamma \vdash t_i \colon T' \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\,r}{\&(\alpha(\widetilde{T}), T_i);\ \mathsf{upd}(\varphi_i, u, b) \wr \Gamma \vdash \mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{T \mapsto \widetilde{\mathsf{upd}(t_i, u, b)}\} \colon T'}$$

where $T \equiv_{eff} T'$, which implies

$$\mathsf{upd}(\&(\alpha(\widetilde{T}), T_n);\ \varphi_n, u, b) \wr \Gamma \vdash \mathsf{upd}(\mathtt{case}\,\alpha(\widetilde{v})\,\mathtt{in}\,\{\widetilde{T \mapsto t'}\}, u, b) \colon T'$$

Case: $t = \mathtt{error}$

This case can never be typed and hence can be disregarded as the hypothesis assumes that t is typed.

Case: $t = f(t_1)$

$$\frac{\varphi \wr \Gamma \vdash t \colon \textsc{Unit}}{f(\varphi) \wr \Gamma \vdash f(t) \colon \textsc{Unit}}$$

We can perform a case split on the relevant conditions in the definition of upd:

Case $b \wedge f \in \mathsf{dom}(u)$

$$\begin{aligned}
\mathsf{upd}(f(t_1), u, \mathtt{true}) &\overset{\text{def}}{=} f(t') \quad (f \mapsto t') \in u \\
\mathsf{upd}(f(\varphi_1), u, \mathtt{true}) &\overset{\text{def}}{=} f(\varphi') \quad (f \mapsto t') \in u \wedge \varphi' \wr \emptyset \vdash t' \colon \gamma
\end{aligned}$$

As we assume updates are well formed then we know that we can make the judgement:

$$\varphi' \wr \emptyset \vdash t' \colon \textsc{Unit}$$

We can then use the TReg typing rule to show that:

$$\frac{\varphi' \wr \Gamma \vdash t' \colon \textsc{Unit}}{f(\varphi') \wr \Gamma \vdash f(t') \colon \textsc{Unit}}$$

and hence that:

$$\mathsf{upd}(f(\varphi_1), u, \mathtt{true}) \wr \Gamma \vdash \mathsf{upd}(f(t_1), u, \mathtt{true}) \colon \textsc{Unit}$$

Case $\neg(b \wedge f \in \mathsf{dom}(u))$

$$\begin{array}{lcll} \mathsf{upd}(f(t_1), u, b) & \overset{\text{def}}{=} & f(\mathsf{upd}(t_1, u, b)) & (\neg b \vee (f \mapsto \gamma) \notin u) \\ \mathsf{upd}(f(\varphi_1), u, b) & \overset{\text{def}}{=} & f(\mathsf{upd}(\varphi, u, b)) & (\neg b \vee (f \mapsto \gamma) \notin u) \end{array}$$

By induction we know that

$$\mathsf{upd}(\varphi_1, u, b) \wr \Gamma \vdash \mathsf{upd}(t_1, u, b) \colon T'$$

where $\textsc{Unit} \equiv_{eff} T'$. The only possible such T' is $\textsc{Unit}$. Hence we can straightforwardly say that:

$$\frac{\mathsf{upd}(\varphi, u, b) \wr \Gamma \vdash \mathsf{upd}(t', u, b) \colon \textsc{Unit}}{f(\mathsf{upd}(\varphi, u, b)) \wr \Gamma \vdash f(\mathsf{upd}(t', u, b)) \colon \textsc{Unit}}$$

Case: $t = \mathtt{dmod}_F(t_1)$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Unit}}{\mathtt{dmod}_\gamma(\varphi_1) \wr \Gamma \vdash \mathtt{dmod}_\gamma(t_1) \colon \textsc{Unit}} \qquad \frac{\varphi \wr \Gamma \vdash t \colon T \qquad \varphi \equiv \varphi'}{\varphi \wr \Gamma \vdash t \colon \varphi'}$$

We can perform a case split on the relevant conditions in the definition of upd:

Case $\widetilde{f} \cap \mathsf{dom}(u) = \emptyset$

By the definition of upd:

$$\begin{array}{lcll} \mathsf{upd}(\mathtt{dmod}_F(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_F(\mathsf{upd}(t_1, u, \mathtt{true})) & \widetilde{f} \cap \mathsf{dom}(u) = \emptyset \\ \mathsf{upd}(\mathtt{dmod}_\gamma(\varphi), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi, u, \mathtt{true})) & \widetilde{f} \cap \mathsf{dom}(u) = \emptyset \end{array}$$

By induction we know that

$$\mathsf{upd}(\varphi_1, u, \mathtt{true}) \wr \Gamma \vdash \mathsf{upd}(t_1, u, \mathtt{true})\colon T'$$

where $\textsc{Unit} \equiv_{eff} T'$. The only possible such T' is $\textsc{Unit}$. Hence we can straightforwardly say that:

$$\frac{\mathsf{upd}(\varphi_1, u, \mathtt{true}) \wr \Gamma \vdash \mathsf{upd}(t_1, u, \mathtt{true})\colon \textsc{Unit}}{\mathtt{dmod}_\gamma(\mathsf{upd}(\varphi_1, u, \mathtt{true})) \wr \Gamma \vdash \mathtt{dmod}_\gamma(\mathsf{upd}(t_1, u, \mathtt{true}))\colon \textsc{Unit}}$$

Case $\widetilde{f} \cap \mathsf{dom}(u) \neq \emptyset$

By the definition of `upd`:

$$\begin{array}{lcll} \mathsf{upd}(\mathtt{dmod}_F(t_1), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_F(\mathsf{upd}(t_1, u, \mathtt{false})) & \widetilde{f} \cap \mathsf{dom}(u) \neq \emptyset \\ \mathsf{upd}(\mathtt{dmod}_\gamma(\varphi), u, b) & \overset{\text{def}}{=} & \mathtt{dmod}_\gamma(\mathsf{upd}(\varphi, u, \mathtt{false})) & \widetilde{f} \cap \mathsf{dom}(u) \neq \emptyset \end{array}$$

By induction we know that

$$\mathsf{upd}(\varphi_1, u, \mathtt{false}) \wr \Gamma \vdash \mathsf{upd}(t_1, u, \mathtt{false})\colon T'$$

where $\textsc{Unit} \equiv_{eff} T'$. The only possible such T' is $\textsc{Unit}$. Hence we can straightforwardly say that:

$$\frac{\mathsf{upd}(\varphi_1, u, \mathtt{false}) \wr \Gamma \vdash \mathsf{upd}(t_1, u, \mathtt{false})\colon \textsc{Unit}}{\mathtt{dmod}_\gamma(\mathsf{upd}(\varphi_1, u, \mathtt{false})) \wr \Gamma \vdash \mathtt{dmod}_\gamma(\mathsf{upd}(t_1, u, \mathtt{false}))\colon \textsc{Unit}}$$

A.6 Thread Update Consistency Lemma

Lemma A.6. *If:*

$$\vdash P \colon \Phi \qquad \forall f \mapsto t.\varphi \wr \emptyset \vdash t \colon \textsc{Unit}$$

then:

$$\vdash \mathsf{upd}(P, u, \mathit{false}) \colon \mathsf{upd}(\Phi, u, \mathit{false})$$

Proof: by induction over P

Case: $P = t$

$$\frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi}$$

By the Expression Update Consistency Lemma:

$$\mathsf{upd}(\varphi, u, b) \wr \emptyset \vdash \mathsf{upd}(t, u, b) \colon T' \qquad T \equiv_{eff} T'$$

Hence:

$$\frac{\mathsf{upd}(\varphi, u, b) \wr \emptyset \vdash \mathsf{upd}(t, u, b) \colon T'}{\vdash \mathsf{upd}(t, u, b) \colon \mathsf{upd}(\varphi, u, b)}$$

Case: $P = P_1 \parallel P_2$

$$\frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2}$$

By induction we know that:

$$\vdash \mathsf{upd}(P_1, u, \mathtt{false}) \colon \mathsf{upd}(\Phi_1, u_i, \mathtt{false}) \qquad \vdash \mathsf{upd}(P_2, u, \mathtt{false}) \colon \mathsf{upd}(\Phi_2, u_i, \mathtt{false})$$

Hence we can say that:

$$\vdash \mathsf{upd}(P_1 \parallel P_2, u, \mathtt{false}) \colon \mathsf{upd}(\Phi_1 \parallel \Phi_2, u, \mathtt{false})$$

A.7 Expression Safety Lemma

Lemma A.7.

$$\varphi \wr \Gamma \vdash t \colon T \Rightarrow \nexists \mathtt{error} \in t$$

Proof. By induction over t

Case $t = n, P = b, P = (), P = r, P = l, P = x, P = X, P = (\alpha(\widetilde{v}), T)$

We trivially know that none of these include `error`.

Case $t = \mathtt{rec}\, X(x : T).t_1$

$$\frac{\begin{array}{c}\varphi_1 \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t_1 \colon T_2 \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi_1) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \mathtt{rec}\, \underline{X}(x : T_1).t_1 \colon T_1 \xrightarrow{\mu \underline{X}.\varphi_1} T_2}$$

We know that $\varphi_1 \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t_1 \colon T_2$. We can then use the inductive hypothesis to show that $\nexists \mathtt{error} \in t_1$. We know straightforwardly that $\nexists \mathtt{error} \in x, T_1, \underline{X}$. Hence we can say that $\nexists \mathtt{error} \in \mathtt{rec}\, \underline{X}(x : T).t_1$.

Case: $t = t_1\, t_1$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \Gamma \vdash t_2 \colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \Gamma \vdash t_1\, t_2 \colon T_1}$$

By induction we know that $\nexists \mathtt{error} \in t_1, t_2$. Hence we have that $\nexists \mathtt{error} \in t_1\, t_1$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \Gamma \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \Gamma \vdash (\alpha(\widetilde{v}), T) \colon T}$$

We know straightforwardly, and by induction, that $\nexists \mathtt{error} \in v_i$. Hence we have that $\nexists \mathtt{error} \in (\alpha(\widetilde{v}), T)$

Case: $t = \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3$

$$\frac{\begin{array}{c}\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Bool} \\ \varphi_2 \wr \Gamma \vdash t_2 \colon T \quad \varphi_3 \wr \Gamma \vdash t_3 \colon T\end{array}}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \Gamma \vdash \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \colon T}$$

By induction we know that $\nexists \texttt{error} \in t_1, t_2, t_3$. Hence we have that $\nexists \texttt{error} \in \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3$.

Case: $t = \texttt{case}\, \alpha(\widetilde{v}) \,\texttt{in}\, \{\widetilde{T_i \mapsto t_i}\}$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \Gamma \vdash \texttt{case}\, \alpha(\widetilde{T}) \,\texttt{in}\, \{\widetilde{T \mapsto t}\} \colon T}$$

We know straightforwardly that $\nexists \texttt{error} \in v_i, T_i$. By induction we know that $\nexists \texttt{error} \in t_i$. Hence we have that $\nexists \texttt{error} \in \texttt{case}\, \alpha(\widetilde{v}) \,\texttt{in}\, \{\widetilde{T_i \mapsto t_i}\}$.

Case: $t = \texttt{let}\, x = \texttt{new}\,\texttt{in}\, t'$

Similar to case $t = \texttt{let}\, x = \alpha(\widetilde{v}) \,\texttt{in}\, t'$.

Case: $t = \texttt{error}$

This case can never be typed and hence can be disregarded as the hypothesis assumes that t is typed.

Case: $t = \texttt{dmod}_{F_1}(t_1)$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi_1) \wr \Gamma \vdash \texttt{dmod}_{\gamma_1}(t_1) \colon \textsc{Unit}}$$

We know straightforwardly that $\nexists \texttt{error} \in \gamma$. By induction we know that $\nexists \texttt{error} \in t_1$. Hence we have that $\nexists \texttt{error} \in \texttt{dmod}_{F_1}(t_1)$.

Case: $t = f(t_1)$

We know straightforwardly that $\nexists$`error` $\in f$. By induction we know that $\nexists$`error` $\in t_1$. Hence we have that $\nexists$`error` $\in f(t_1)$.

$\square$

A.8 Safety Theorem

Theorem A.8.

$$\vdash P \colon \Phi \Rightarrow \nexists \mathit{error} \in P$$

Proof. By induction over P

Case $P = t$

$$\frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi}$$

As $\varphi \wr \emptyset \vdash t \colon T$, by Lemma A.7 we know that $\nexists\texttt{error} \in t$

Case $P = P_1 \parallel P_2$

$$\frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2}$$

By induction we know that $\nexists\texttt{error} \in P_1, P_2$. Hence we have that $\nexists\texttt{error} \in P_1 \parallel P_2$

□

A.9 Value Effect Lemma

Lemma A.9.

$$t = v \Rightarrow \epsilon \wr \Gamma \vdash t \colon T$$

Proof. Proof: by case split over v

Case $v = n$

$$\frac{}{\epsilon \wr \Gamma \vdash n \colon \textsc{Int}}$$

Case $v = b$

$$\frac{}{\epsilon \wr \Gamma \vdash b \colon \textsc{Bool}}$$

Case $v = ()$

$$\frac{}{\epsilon \wr \Gamma \vdash () \colon \textsc{Unit}}$$

Case $v = r$

$$\frac{}{\epsilon \wr \Gamma \vdash r \colon \mathsf{Res}\, r}$$

Case $v = x$

$$\frac{}{\epsilon \wr \Gamma \vdash x \colon \Gamma(x)}$$

Case $v = l$

$$\frac{}{\epsilon \wr \Gamma \vdash l \colon \mathtt{Lab}\, l}$$

Case $v = \texttt{rec}\, X(x : T).t_1$

$$\frac{\begin{array}{c}\varphi \wr \Gamma, x \colon T_1, \underline{X} \colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t \colon T_2 \\ x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \Gamma \vdash \texttt{rec}\, \underline{X}(x : T_1).t \colon T_1 \xrightarrow{\mu \underline{X}.\varphi} T_2}$$

□

A.10 General Compatibility Well Definedness Lemma

Lemma A.10. *$\mathcal{G}(\Sigma, \Phi, \psi, \omega)$ is well defined fixed point satisfying $\bigwedge_{k \in \mathcal{N}} \mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$.*

$\mathcal{G}(\Sigma, \Phi, \psi, \omega) = \bigwedge_{k \in \mathcal{N}} \mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$:

$$\mathcal{G}^0(\Sigma, \Phi, \psi, \omega) = \texttt{true}$$

$$\begin{array}{l}
\mathcal{G}^k(\Sigma, \Phi, \psi, \omega) = \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi) \wedge \\
\qquad (\forall \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \Rightarrow \mathcal{G}^{k-1}(\Sigma', \Phi', \psi, \omega)) \\
\qquad \wedge (\exists \Sigma', \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \ \vee\ \Phi = \Pi_I\, \epsilon) \\
\vee \psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n \ \wedge\ i \text{ smallest in 1...n.} \\
\qquad p_i(\Sigma, \Phi)\ \wedge\ \mathcal{G}^{k-1}(\Sigma, \mathsf{upd}(\Phi, u_i, \texttt{false}), \psi \setminus p_i \mapsto u_i, \omega)
\end{array}\right] \\
\wedge \nexists \texttt{error} \in \Phi\ \wedge\ \mathcal{G}^{k-1}(\Sigma, \Phi, (\psi, \omega), \emptyset)
\end{array}\right.
\end{array}$$

where $k > 0$

Proof: We prove (anti) monotonicity by induction over k: if $\mathcal{G}^{k+1}(\Sigma, \Phi, \psi, \omega)$ then $\mathcal{G}^k(\Sigma, \Phi, \psi, \omega)$.

Case $k = 0$

By the definition $\mathcal{G}^0(\Sigma, \Phi, \psi, \psi') = \texttt{true}$, and hence we trivially have that given $\mathcal{G}^1(\Sigma, \Phi, \psi, \omega)$ then $\mathcal{G}^0(\Sigma, \Phi, \psi, \omega)$.

Case $k = n > 0$

By the hypothesis we know that:

$$\left[\begin{array}{l}
\left[\begin{array}{l}
\nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi) \wedge \\
\qquad (\forall \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \Rightarrow \mathcal{G}^{n}(\Sigma', \Phi', \psi, \omega)) \\
\qquad \wedge (\exists \Sigma', \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \ \vee\ \Phi = \Pi_I\, \epsilon) \\
\vee \psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n \ \wedge\ i \text{ smallest in 1...n.} \\
\qquad p_i(\Sigma, \Phi)\ \wedge\ \mathcal{G}^{n}(\Sigma, \mathsf{upd}(\Phi, u_i, \texttt{false}), \psi \setminus p_i \mapsto u_i, \omega)
\end{array}\right] \\
\wedge \nexists \texttt{error} \in \Phi\ \wedge\ \mathcal{G}^{n}(\Sigma, \Phi, (\psi, \omega), \emptyset)
\end{array}\right.$$

holds.

By induction we know that if $k = n-1$ that $\mathcal{G}^n(\Sigma, \Phi, \psi, \omega)$ then $\mathcal{G}^{n-1}(\Sigma, \Phi, \psi, \omega)$. Hence we can show that:

$$\left[\begin{array}{l}
\left[\begin{array}{l}
\nexists p \in \mathsf{dom}(\psi).p(\Sigma, \Phi) \wedge \\
\qquad (\forall \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \Rightarrow \mathcal{G}^{n-1}(\Sigma', \Phi', \psi, \omega)) \\
\qquad \wedge\, (\exists \Sigma', \Phi'.[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi' \,\vee\, \Phi = \Pi_I\, \epsilon) \\
\vee \psi = p_1 \mapsto u_1, \ldots, p_n \mapsto u_n \,\wedge\, i \text{ smallest in 1...n.} \\
\qquad p_i(\Sigma, \Phi) \,\wedge\, \mathcal{G}^{n-1}(\Sigma, \mathsf{upd}(\Phi, u_i, \texttt{false}), \psi \setminus p_i \mapsto u_i, \omega)
\end{array}\right] \\
\wedge\, \nexists \texttt{error} \in \Phi \,\wedge\, \mathcal{G}^{n-1}(\Sigma, \Phi, (\psi, \omega), \emptyset)
\end{array}\right.$$

which, as $n > 0$, is the definition of $\mathcal{G}^n(\Sigma, \Phi, \psi, \omega)$.

A.11 Expression Subject Reduction Lemma

Lemma 3.1

If:

$$\varphi \wr \emptyset \vdash t\colon T \qquad [\Sigma]\,\varphi \rightarrow [\Sigma']\,\varphi' \Rightarrow \nexists\texttt{error} \in \varphi' \qquad [\sigma]\,t \xrightarrow{\gamma}_F [\sigma']\,t'$$

Then

$$\varphi' \wr \emptyset \vdash t'\colon T \qquad [\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \gamma' \vdash \gamma$$

Proof. By induction over $[\sigma]\,t \rightarrow [\sigma']\,t'$.

Case RAppOne

$$\frac{\varphi_1 \wr \emptyset \vdash t_1\colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \emptyset \vdash t_2\colon T_2}{(\varphi_1;\,\varphi_2;\,\varphi_3) \wr \emptyset \vdash t_1\,t_2\colon T_1}$$

By induction we know that if:

$$[\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \texttt{error} \notin \varphi_1' \qquad [\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'$$

then:

$$\varphi_1' \wr \emptyset \vdash t_1'\colon T \qquad \gamma' \vdash \gamma$$

Hence we can show that:

$$\frac{[\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'}{[\sigma]\,t_1\,t_2 \xrightarrow{\gamma}_F [\sigma']\,t_1'\,t_2}$$

Case RAppTwo

Similar to case RAppOne

Case RAppThree

$$\frac{}{[\sigma]\,\texttt{rec}\,\underline{X}(x:T).t\,v \xrightarrow{\tau} [\sigma]\,t[\texttt{rec}\,\underline{X}(x:T).t/\underline{X}][v/x]}$$

$$\frac{\epsilon \wr \emptyset \vdash \texttt{rec}\,\underline{X}(x:T).t\colon T_2 \xrightarrow{\mu\underline{X}.\varphi_3} T_1 \qquad \epsilon \wr \emptyset \vdash v\colon T_2}{(\epsilon;\,\epsilon;\,\mu\underline{X}.\varphi_3) \wr \emptyset \vdash \texttt{rec}\,\underline{X}(x:T).t\,v\colon T_1}$$

$$\frac{\begin{array}{c}\varphi \wr \emptyset, x\colon T_1, \underline{X}\colon T_1 \xrightarrow{\underline{X}} T_2 \vdash t\colon T_2 \\ x \notin \mathsf{fv}(\emptyset), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi) \qquad \underline{X} \notin \mathsf{fv}(T_1), \mathsf{fv}(T_2)\end{array}}{\epsilon \wr \emptyset \vdash \texttt{rec}\,\underline{X}(x:T_1).t\colon T_1 \xrightarrow{\mu\underline{X}.\varphi} T_2}$$

Hence we know that:

$$\mu \underline{X}.\varphi_3 \wr \emptyset \vdash \texttt{rec}\, \underline{X}(x : T).t\, v \colon T_1$$

By Lemma A.2 and Lemma A.3 we know that:

$$\varphi_3[\mu\underline{X}.\varphi_3/\underline{X}][v/x] \wr \emptyset[v/x] \vdash t[\texttt{rec}\, \underline{X}(x : T).t/\underline{X}][v/x] \colon T_2[v/x][\mu\underline{X}.\varphi_3/\underline{X}]$$

as $x \notin \mathsf{fv}(\Gamma), \mathsf{fv}(T_1), \mathsf{fv}(T_2), \mathsf{fv}(\varphi_3)$ and $\underline{X} \notin \mathsf{fv}(T_2)$ we can say that:

$$\varphi_3[\mu\underline{X}.\varphi_3/\underline{X}] \wr \emptyset \vdash t[\texttt{rec}\, \underline{X}(x : T).t/\underline{X}][v/x] \colon T_2$$

As

$$\varphi_3[\mu\underline{X}.\varphi_3/\underline{X}] \equiv \varphi_3$$

we can use RTREC to show that:

$$\frac{}{[\Sigma]\, \mu\underline{X}.\varphi_3 \xrightarrow{\tau} [\Sigma]\, \varphi_3[\mu\underline{X}.\varphi_3/\underline{X}]}$$

where $\gamma = \tau$.

Case RAccOne

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T}{[\sigma]\, (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\, \sigma(\alpha(\widetilde{v}))} \qquad \frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By our assumptions we know that $\sigma(\alpha(\widetilde{v})) = v$. By Lemma A.9 we know that:

$$t = v \Rightarrow \epsilon \wr \Gamma \vdash t \colon T$$

We can show that $[\Lambda(\sigma)]\, ((\alpha(\widetilde{v}), T), T) \to [\Lambda(\sigma')]\, \epsilon$ using the RTACCSUCC rule:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\, (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\, \epsilon}$$

According to TACC we know that:

$$\epsilon \wr \Gamma \vdash v_i \colon T_i$$

and hence we can show that:

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i}{\alpha(\widetilde{T}) \vdash \alpha(\widetilde{v})}$$

Case RAccTwo

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v}))\colon T' \neq T}{[\sigma]\,(\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\,\texttt{error}} \qquad \frac{\epsilon \wr \emptyset \vdash v_i\colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T)\colon T}$$

Using the premises of RAccTwo we can show that:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) \neq T}{[\Sigma]\,(\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\,\texttt{error}}$$

By the premises we know that:

$$[\Lambda(\sigma)]\,(\alpha(\widetilde{T}), T) \to [\Lambda(\sigma')]\,\varphi'' \Rightarrow \nexists\texttt{error} \in \varphi''$$

This is a contradiction. Hence this reduction cannot occur, and we can disregard it.

Case RIfOne

$$\frac{}{[\sigma]\,\texttt{if}\,\texttt{true}\,\texttt{then}\,t_2\,\texttt{else}\,t_3 \xrightarrow{\tau} [\sigma]\,t_2}$$

$$\frac{\begin{array}{c}\varphi_1 \wr \emptyset \vdash \texttt{true}\colon \textsc{Bool} \\ \varphi_2 \wr \emptyset \vdash t_2\colon T \quad \varphi_3 \wr \emptyset \vdash t_3\colon T\end{array}}{\varphi_1;\ (\varphi_2 \oplus \varphi_3) \wr \emptyset \vdash \texttt{if}\,\texttt{true}\,\texttt{then}\,t_2\,\texttt{else}\,t_3\colon T}$$

By Lemma A.9 we know that $\varphi_1 = \epsilon$, and hence that:

$$\varphi_2 \oplus \varphi_3 \wr \Gamma \vdash \texttt{if}\,\texttt{true}\,\texttt{then}\,t_2\,\texttt{else}\,t_3\colon T$$

By TInt we know that:

$$\varphi_2 \wr \emptyset \vdash t_2\colon T$$

We can show that $[\Lambda(\sigma)]\,\varphi_2 \oplus \varphi_3 \xrightarrow{\gamma'} [\Lambda(\sigma)]\,\varphi_2$ using RTIntChoice:

$$\frac{i \in 1, 2}{[\Sigma]\,\varphi_1 \oplus \varphi_2 \xrightarrow{\tau} [\Sigma]\,\varphi_i}$$

Case RIfTwo

Similar to case RIfOne.

Case RIfThree

$$\frac{\begin{array}{c}\varphi_1 \wr \emptyset \vdash t_1 \colon \textsc{Bool} \\ \varphi_2 \wr \emptyset \vdash t_2 \colon T \quad \varphi_3 \wr \emptyset \vdash t_3 \colon T\end{array}}{\varphi_1;\ (\varphi_2 \oplus \varphi_3) \wr \emptyset \vdash \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \colon T}$$

By induction we know that if:

$$[\Lambda(\sigma)]\, \varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\, \varphi' \qquad \texttt{error} \notin \varphi_1' \qquad [\sigma]\, t_1 \xrightarrow{\gamma}_F [\sigma']\, t_1'$$

then:

$$\varphi_1' \wr \emptyset \vdash t_1' \colon T \qquad \gamma' \vdash \gamma$$

Hence we can show that:

$$\frac{[\sigma']\, t_1 \xrightarrow{\gamma}_F [\sigma]\, t_1'}{[\sigma]\, \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \xrightarrow{\gamma}_F [\sigma']\, \texttt{if}\, t_1'\, \texttt{then}\, t_2\, \texttt{else}\, t_3}$$

Case RCaseOne

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T_i \qquad T_i \in \widetilde{T}}{[\sigma]\, \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\, t_i} \qquad \frac{\varphi_i \wr \emptyset \vdash t_i \colon T \qquad \epsilon \wr \emptyset \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \emptyset \vdash \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T \mapsto t}\} \colon T}$$

By TCase we know that $\varphi_i \wr \emptyset \vdash t_i \colon T$. As $\sigma(l)\colon T_i$ we can use RTAccSucc to show that:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\, (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\, \epsilon}$$

We can then use RTExtChoiceOne and RTExtChoiceTwo to show that:

$$[\Lambda(\sigma)]\, \&(\alpha(\widetilde{T}), T);\ \varphi_i \to [\Lambda(\sigma'')]\, \varphi_i$$

By our assumptions we know that if:

$$\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \alpha(\widetilde{T}) \vdash \alpha(\widetilde{v})$$

that

$$\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma')$$

By TCase we know that $\epsilon \wr \emptyset \vdash v_i \colon T_i$, and hence $\sigma'' = \sigma'$.

Case RCaseTwo

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v}))\colon T' \qquad T' \notin \widetilde{T}}{[\sigma]\,\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\,\texttt{error}} \qquad \frac{\varphi_i \wr \emptyset \vdash t_i\colon T \qquad \epsilon \wr \emptyset \vdash v_i\colon T_i \qquad T_i \neq \mathsf{Res}\,r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \emptyset \vdash \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}\colon T}$$

As $T' \notin \widetilde{T}$, by RTAccErr we know that $\forall T_i \in \widetilde{T}$:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) \neq T}{[\Sigma]\,(\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\,\texttt{error}}$$

Hence we can use

$$\frac{\forall i\,.\,([\Sigma]\,\varphi_i \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma_1]\,\varphi_i' \,\wedge\, \texttt{error} \in \varphi_i')}{[\Sigma]\,\varphi_1 \& \varphi_2 \xrightarrow{\alpha(\widetilde{T})_j}_F [\Sigma_i]\,\varphi_j'}$$

to show that:

$$[\Lambda(\sigma)]\,\&(\alpha(\widetilde{T}), T);\, \varphi_i \rightarrow [\Lambda(\sigma')]\,\texttt{error};\, \varphi_i$$

By the hypotheses we know that:

$$[\Lambda(\sigma)]\,(\alpha(\widetilde{T}), T) \rightarrow [\Lambda(\sigma')]\,\varphi'' \Rightarrow \nexists \texttt{error} \in \varphi''$$

This is a contradiction. Hence this reduction cannot occur, and we can disregard it.

Case RRegOne

$$\frac{\varphi_1 \wr \Gamma \vdash t_1\colon \textsc{Unit}}{f(\varphi_1) \wr \Gamma \vdash f(t_1)\colon \textsc{Unit}}$$

By induction we know that if:

$$[\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \texttt{error} \notin \varphi_1' \qquad [\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'$$

then:

$$\varphi_1' \wr \emptyset \vdash t_1'\colon T \qquad \gamma' \vdash \gamma$$

Hence we can show that:

$$\frac{[\sigma]\,t_1 \xrightarrow{\tau}_F [\sigma']\,t_1'}{[\sigma']\,f(t_1) \xrightarrow{\tau}_F [\sigma']\,f(t_1')}$$

Case RRegTwo

$$\frac{\varphi_1 \wr \Gamma \vdash t_1\colon \textsc{Unit}}{f(\varphi_1) \wr \Gamma \vdash f(t_1)\colon \textsc{Unit}}$$

By induction we know that if:

$$[\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \texttt{error} \notin \varphi_1' \qquad [\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'$$

then:

$$\varphi_1' \wr \emptyset \vdash t_1' \colon T \qquad \gamma' \vdash \gamma$$

Hence we can show that:

$$\frac{[\sigma]\,t \xrightarrow{\alpha(\widetilde{v})}_F [\sigma']\,t'}{[\sigma]\,f(t) \xrightarrow{\alpha(\widetilde{v})}_{F\cup\{f\}} [\sigma']\,t'}$$

Case RRegThree

$$\frac{}{[\sigma]\,f(t) \xrightarrow{\tau} [\sigma]\,t} \qquad \frac{\varphi_1 \wr \Gamma \vdash v \colon \textsc{Unit}}{f(\varphi_1) \wr \Gamma \vdash f(v) \colon \textsc{Unit}}$$

By Lemma A.9 we know that $\varphi_1 = \epsilon$. We can straightforwardly show that:

$$\frac{}{[\Sigma]\,f(\epsilon) \xrightarrow{\tau} [\Sigma]\,\epsilon}$$

Case RDmodOne

$$\frac{\varphi \wr \Gamma \vdash t \colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi) \wr \Gamma \vdash \texttt{dmod}_{\gamma_1}(t) \colon \textsc{Unit}}$$

By induction we know that if:

$$[\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \texttt{error} \notin \varphi_1' \qquad [\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'$$

then:

$$\varphi_1' \wr \emptyset \vdash t_1' \colon T \qquad \gamma' \vdash \gamma$$

Hence we can show that:

$$\frac{[\sigma]\,t \xrightarrow{\gamma}_{F'} [\sigma']\,t' \qquad \texttt{regions}(t) \not\subseteq F}{[\sigma]\,\texttt{dmod}_F(t) \xrightarrow{\gamma}_{F'} [\sigma']\,\texttt{dmod}_{F\cup F'}(t')}$$

Case $\textbf{\scriptsize RDmodTwo}$

$$\frac{\texttt{regions}(t) \subseteq F}{[\sigma]\,\texttt{dmod}_F(t) \xrightarrow{\mathcal{T}} [\sigma]\,t} \qquad \frac{\varphi \wr \Gamma \vdash t\colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi) \wr \Gamma \vdash \texttt{dmod}_{\gamma_1}(t)\colon \textsc{Unit}}$$

We can straightforwardly use $\textsc{rtDmodTwo}$ to show that:

$$\frac{\texttt{regions}(\varphi_1) \subseteq F}{[\Lambda(\sigma)]\,\texttt{dmod}_F(\varphi_1) \xrightarrow{\mathcal{T}} [\Lambda(\sigma)]\,\varphi_1}$$

$\square$

A.12 Process Subject Reduction Lemma

Lemma 3.2

If:

$$\vdash P \colon \Phi \qquad [\Sigma]\,\Phi \rightarrow [\Sigma']\,\Phi' \Rightarrow \nexists \texttt{error} \in \Phi' \qquad [\sigma]\,P \xrightarrow{\gamma} [\sigma']\,P'$$

then

$$\vdash P' \colon \Phi' \qquad [\Lambda(\sigma)]\,\Phi \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\Phi'$$

where

$$\gamma' \vdash \gamma$$

Proof. By induction over P

Case: $P = t$

This case can be typed using:

$$\frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi} \qquad \frac{\vdash t \colon \varphi \qquad \texttt{res}(t) \cup \texttt{res}(\sigma) \subseteq R \quad \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma), \varphi, \psi, \omega) \text{ and } (\mathcal{C} \implies \mathcal{S})}{\vdash^{\mathcal{S}}_{\omega} [\sigma]\,t, \psi \colon \varphi}$$

Using Lemma 3.1 we know that, if:

$$\varphi \wr \emptyset \vdash t \colon T \qquad \mathcal{S}(\Sigma, \varphi, \psi, \omega) \qquad [\sigma]\,t \xrightarrow{\gamma}_F [\sigma']\,t'$$

then

$$\varphi' \wr \emptyset \vdash t' \colon T \qquad [\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'}_F [\Lambda(\sigma')]\,\varphi' \qquad \gamma' \vdash \gamma$$

We can then apply the TThread typing rule:

$$\frac{\varphi' \wr \emptyset \vdash t' \colon T}{\vdash t' \colon \varphi'}$$

Case: $P = P_1 \parallel P_2$

This case can be typed using:

$$\frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2} \qquad \frac{\begin{array}{c}\vdash t \colon \varphi \qquad \texttt{res}(t) \cup \texttt{res}(\sigma) \subseteq R \\ \exists \mathcal{C}.\mathcal{C}(\Lambda(\sigma), \varphi, \psi, \omega) \text{ and } (\mathcal{C} \implies \mathcal{S})\end{array}}{\vdash^{\mathcal{S}}_{\omega} [\sigma]\, t, \psi \colon \varphi}$$

By the reduction rules:

$$\frac{[\Sigma]\, \Phi_1 \xrightarrow{\gamma}_F [\Sigma']\, \Phi_1'}{[\Sigma]\, \Phi_1 \parallel \Phi_2 \xrightarrow{\gamma}_F [\Sigma']\, \Phi_1' \parallel \Phi_2} \qquad \frac{[\Sigma]\, \Phi_2 \xrightarrow{\gamma}_F [\Sigma']\, \Phi_2'}{[\Sigma]\, \Phi_1 \parallel \Phi_2 \xrightarrow{\gamma}_F [\Sigma']\, \Phi_1 \parallel \Phi_2'}$$

we know that if:

$$\nexists \texttt{error} \in \Phi_1 \parallel \Phi_2 \qquad [\Sigma]\, \Phi_1 \parallel \Phi_2 \rightarrow [\Sigma']\, \Phi' \Rightarrow \nexists \texttt{error} \in \varphi'$$

then:

$$\nexists \texttt{error} \in \Phi_1, \Phi_2 \qquad [\Sigma]\, \Phi_1 \rightarrow [\Sigma']\, \Phi_1' \Rightarrow \nexists \texttt{error} \in \varphi' \qquad [\Sigma]\, \Phi_2 \rightarrow [\Sigma']\, \Phi_2' \Rightarrow \nexists \texttt{error} \in \varphi'$$

We can then use the inductive hypothesis to show that if:

$$[\sigma]\, P_1 \xrightarrow{\gamma} [\sigma']\, P_1'$$

then

$$\vdash P_1' \colon \Phi_1' \qquad [\Lambda(\sigma)]\, \Phi \xrightarrow{\gamma'} [\Lambda(\sigma')]\, \Phi'$$

We can then show that:

$$\frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2}$$

and:

$$\frac{[\Lambda(\sigma)]\, \Phi_1 \xrightarrow{\gamma}_F [\Lambda(\sigma')]\, \Phi_1'}{[\Lambda(\sigma)]\, \Phi_1 \parallel \Phi_2 \xrightarrow{\gamma}_F [\Lambda(\sigma')]\, \Phi_1' \parallel \Phi_2}$$

The case for P_2 is symmetrical.

□

A.13 Empty Effect Expression Liveness Lemma

Lemma 3.3

If

$$\epsilon \wr \emptyset \vdash t \colon T$$

then

$$[\sigma]\, t \xrightarrow{\gamma'} [\sigma]\, t' \;\vee\; t = v$$

Proof. By induction over t.

Case $t = v$

We trivially have that $t = v$.

Case: $t = t_1\, t_2$

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \emptyset \vdash t_2 \colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \emptyset \vdash t_1\, t_2 \colon T_1}$$

We perform a case split on the structure of $t_1\, t_2$

Case $t_1 \neq v$

By the hypothesis and the TApp rule we know that:

$$\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad [\Sigma]\; \varphi \to [\Sigma']\; \varphi' \qquad \varphi \equiv \varphi_1;\, \varphi_2;\, \varphi_3$$

By the inductive hypothesis we know that:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma]\, t_1' \;\vee\; t_1 = v$$

We know the latter cannot be the case, as it contradicts the assumption that $t_1 \neq v$. Hence:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma]\, t_1'$$

Hence we can show that:

$$\frac{[\sigma]\, t_1 \xrightarrow{\gamma}_F [\sigma']\, t_1'}{[\sigma]\, t_1\, t_2 \xrightarrow{\gamma}_F [\sigma']\, t_1'\, t_2}$$

Case $t_1 = v, t_2 \neq v'$

By the hypothesis and the tApp rule we know that:

$$\varphi_2 \wr \emptyset \vdash t_2 \colon T_2$$

By the inductive hypothesis we know that:

$$[\sigma]\, t_2 \xrightarrow{\gamma'} [\sigma]\, t_2' \,\vee\, t_2 = v$$

We know the latter cannot be the case, as it contradicts the assumption that $t_2 \neq v$. Hence:

$$[\sigma]\, t_2 \xrightarrow{\gamma'} [\sigma]\, t_2'$$

Hence we can show that:

$$\frac{[\sigma]\, t_2 \xrightarrow{\gamma}_F [\sigma']\, t_2'}{[\sigma]\, v\, t_2 \xrightarrow{\gamma}_F [\sigma']\, v\, t_2'}$$

Case $t_1 = v, t_2 = v'$

As $t_1 = v$ then we know that $t_1 = \texttt{rec}\, X(x : T).t_3$. Hence we can straightforwardly apply the rAppThree rule:

$$\frac{}{[\sigma]\, \texttt{rec}\, \underline{X}(x : T).t\, v \xrightarrow{\tau} [\sigma]\, t[\texttt{rec}\, \underline{X}(x : T).t/\underline{X}][v/x]}$$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By the hypothesis we know that

$$(\alpha(\widetilde{v}), T) \equiv \epsilon$$

This is a contradiction, and hence we can disregard this case.

Case: $t = \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3$

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon \textsc{Bool} \qquad \varphi_2 \wr \emptyset \vdash t_2 \colon T \quad \varphi_3 \wr \emptyset \vdash t_3 \colon T}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \emptyset \vdash \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \colon T}$$

By the inductive hypothesis we know that:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma]\, t_1' \;\vee\; t_1 = v$$

If the former is the case then we can show that:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma]\, t_1'$$

and hence that:

$$\frac{[\sigma']\, t_1 \xrightarrow{\gamma}_F [\sigma]\, t_1'}{[\sigma]\, \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \xrightarrow{\gamma}_F [\sigma']\, \texttt{if}\, t_1'\, \texttt{then}\, t_2\, \texttt{else}\, t_3}$$

If the latter is the case then we can show that:

$$\frac{}{[\sigma]\, \texttt{if}\, \texttt{true}\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \xrightarrow{\tau} [\sigma]\, t_2} \qquad \frac{}{[\sigma]\, \texttt{if}\, \texttt{false}\, \texttt{then}\, t_2\, \texttt{else}\, t_3 \xrightarrow{\tau} [\sigma]\, t_3}$$

Case: $t = \texttt{case}\, \alpha(\widetilde{T})\, \texttt{in}\, \{\widetilde{T \mapsto t}\}$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\, r}{\&(\alpha(\widetilde{T}), T_i);\, \varphi_i \wr \Gamma \vdash \texttt{case}\, \alpha(\widetilde{v})\, \texttt{in}\, \{\widetilde{T \mapsto t}\} \colon T}$$

By the hypothesis we know that

$$\&(\alpha(\widetilde{T}), T);\, \varphi_i \equiv \epsilon$$

This is a contradiction, and hence we can disregard this case.

Case: $t = f(t_1)$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Unit}}{f(\varphi_1) \wr \Gamma \vdash f(t_1) \colon \textsc{Unit}}$$

By the hypothesis we know that

$$f(\varphi_1) \equiv \epsilon$$

This is a contradiction, and hence we can disregard this case.

Case: $t = \texttt{dmod}_{F_1}(t_1)$

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi_1) \wr \emptyset \vdash \texttt{dmod}_{\gamma_1}(t_1) \colon \textsc{Unit}}$$

By the hypothesis we know that

$$\texttt{dmod}_F(\varphi_1) \equiv \epsilon$$

This is a contradiction, and hence we can disregard this case. $\square$

A.14 Active Effect Expression Liveness Lemma

Lemma 3.4

If

$$\varphi \wr \emptyset \vdash t \colon T \qquad [\Lambda(\sigma)]\,\varphi \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\varphi'$$

then

$$[\sigma]\,t \xrightarrow{\gamma'} [\sigma']\,t'$$

Proof. By induction over t.

Case $t = v$

By Lemma A.9 we know that $\varphi = \epsilon$. There are no reduction rules for ϵ. Therefore the premises of the lemma are not fulfilled, and we can disregard the case.

Case: $t = t_1\,t_2$

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \varphi_2 \wr \emptyset \vdash t_2 \colon T_2}{(\varphi_1;\, \varphi_2;\, \varphi_3) \wr \emptyset \vdash t_1\,t_2 \colon T_1}$$

We perform a case split on the structure of $t_1\,t_2$

Case $t_1 \neq v$

By the hypothesis and the TAPP rule we know that:

$$\varphi_1 \wr \emptyset \vdash t_1 \colon T_2 \xrightarrow{\varphi_3} T_1 \qquad \boxed{[\Sigma]}\,\varphi \to \boxed{[\Sigma']}\,\varphi' \qquad \varphi \equiv \varphi_1;\, \varphi_2;\, \varphi_3$$

Either $\varphi_1 \equiv \epsilon$ or $\varphi_1 \not\equiv \epsilon$.

Case: $\varphi_1 \equiv \epsilon$ By Lemma 3.3 we know that:

$$[\sigma]\,t_1 \xrightarrow{\gamma'} [\sigma']\,t_1' \lor t_1 = v$$

We know the latter cannot be the case, as it contradicts the assumption that $t_1 \neq v$. Hence:

$$[\sigma]\,t_1 \xrightarrow{\gamma'} [\sigma']\,t_1'$$

Case: $\varphi_1 \not\equiv \epsilon$ The reduction rule for sequencing is:

$$\frac{[\Sigma]\,\varphi_1 \xrightarrow{\gamma}_F [\Sigma']\,\varphi_1'}{[\Sigma]\,\varphi_1;\ \varphi_2;\ \varphi_3 \xrightarrow{\gamma}_F [\Sigma']\,\varphi_1';\ \varphi_2;\ \varphi_3}$$

By this reduction rule we know that:

$$[\Lambda(\sigma)]\,\varphi_1 \xrightarrow{\gamma} [\Lambda(\sigma')]\,\varphi_1'$$

Hence we can use the inductive hypothesis to show that:

$$[\sigma]\,t_1 \xrightarrow{\gamma} [\sigma']\,t_1'$$

Hence we can show that, irrespective of whether or not $\varphi_1 \equiv \epsilon$ that:

$$\frac{[\sigma]\,t_1 \xrightarrow{\gamma}_F [\sigma']\,t_1'}{[\sigma]\,t_1\,t_2 \xrightarrow{\gamma}_F [\sigma']\,t_1'\,t_2}$$

Case $t_1 = v, t_2 \neq v'$

By the hypothesis and the TApp rule we know that:

$$\varphi_2 \wr \emptyset \vdash t_2 : T_2$$

By Lemma A.9, we know that $\varphi_1 = \epsilon$. Either $\varphi_2 \equiv \epsilon$ or $\varphi_2 \not\equiv \epsilon$.

Case: $\varphi_2 \equiv \epsilon$ By Lemma 3.3 we know that:

$$[\sigma]\,t_2 \xrightarrow{\gamma'} [\sigma']\,t_2' \lor t_2 = v$$

We know the latter cannot be the case, as it contradicts the assumption that $t_2 \neq v$. Hence:

$$[\sigma]\,t_2 \xrightarrow{\gamma'} [\sigma']\,t_2'$$

Case: $\varphi_2 \not\equiv \epsilon$ The reduction rule for sequencing is:

$$\frac{[\Sigma]\,\varphi_2 \xrightarrow{\gamma}_F [\Sigma']\,\varphi_2'}{[\Sigma]\,\varphi_2;\ \varphi_3 \xrightarrow{\gamma}_F [\Sigma']\,\varphi_2';\ \varphi_3}$$

By this reduction rule we know that:

$$[\Lambda(\sigma)]\,\varphi_1 \xrightarrow{\gamma} [\Lambda(\sigma')]\,\varphi_1'$$

Hence we can use the inductive hypothesis to show that:

$$[\sigma]\, t_1 \xrightarrow{\gamma} [\sigma']\, t_1'$$

Hence we can show that, irrespective of whether or not $\varphi_2 \equiv \epsilon$ that:

$$\frac{[\sigma]\, t_2 \xrightarrow{\gamma}_F [\sigma']\, t_2'}{[\sigma]\, v\, t_2 \xrightarrow{\gamma}_F [\sigma']\, v\, t_2'}$$

Case $t_1 = v, t_2 = v'$

As $t_1 = v$ then we know that $t_1 = \mathtt{rec}\, X(x : T).t_3$. Hence we can straightforwardly apply the RAppThree rule:

$$\frac{}{[\sigma]\, \mathtt{rec}\, \underline{X}(x : T).t\, v \xrightarrow{\tau} [\sigma]\, t[\mathtt{rec}\, \underline{X}(x : T).t/\underline{X}][v/x]}$$

Case: $t = (\alpha(\widetilde{v}), T)$

$$\frac{\epsilon \wr \emptyset \vdash v_i \colon T_i \qquad \mathsf{Res}\, r \notin T}{(\alpha(\widetilde{T}), T) \wr \emptyset \vdash (\alpha(\widetilde{v}), T) \colon T}$$

By the hypothesis we know that

$$[\Sigma]\ (\alpha(\widetilde{T}), T) \to [\Sigma']\ \varphi'$$

The reduction rules for $(\alpha(\widetilde{T}), T)$ that fulfil these conditions are:

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\, (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\, \epsilon} \qquad \frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) \neq T}{[\Sigma]\, (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\, \mathtt{error}}$$

Hence we can show that $\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma')$. By our assumptions we know that:

$$\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \Leftrightarrow \sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$$

Hence we can reduce $(\alpha(\widetilde{v}), T)$ using either the RAccSucc or RAccErr rule:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T}{[\sigma]\, (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\, \sigma(\alpha(\widetilde{v}))} \qquad \frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \quad \sigma(\alpha(\widetilde{v})) \colon T' \neq T}{[\sigma]\, (\alpha(\widetilde{v}), T) \xrightarrow{\alpha(\widetilde{v})} [\sigma']\, \mathtt{error}}$$

Case: $t = \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3$

$$\frac{\begin{array}{c}\varphi_1 \wr \emptyset \vdash t_1 \colon \textsc{Bool} \\ \varphi_2 \wr \emptyset \vdash t_2 \colon T \quad \varphi_3 \wr \emptyset \vdash t_3 \colon T\end{array}}{\varphi_1;\, (\varphi_2 \oplus \varphi_3) \wr \emptyset \vdash \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \colon T}$$

Either $\varphi_1 \equiv \epsilon$ or $\varphi_1 \not\equiv \epsilon$.

Case: $\varphi_1 \equiv \epsilon$ By Lemma 3.3 we know that:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma']\, t_1' \vee t_1 = v$$

If the former is the case then we can show that:

$$[\sigma]\, t_1 \xrightarrow{\gamma'} [\sigma']\, t_1'$$

and hence that:

$$\frac{[\sigma']\, t_1 \xrightarrow{\gamma}_F [\sigma]\, t_1'}{[\sigma]\, \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \xrightarrow{\gamma}_F [\sigma']\, \texttt{if}\, t_1' \,\texttt{then}\, t_2 \,\texttt{else}\, t_3}$$

If the latter is the case then we can show that:

$$\frac{}{[\sigma]\, \texttt{if}\,\texttt{true}\,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \xrightarrow{\tau} [\sigma]\, t_2} \qquad \frac{}{[\sigma]\, \texttt{if}\,\texttt{false}\,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \xrightarrow{\tau} [\sigma]\, t_3}$$

Case: $\varphi_1 \not\equiv \epsilon$ The reduction rule for sequencing is:

$$\frac{[\Sigma]\, \varphi_1 \xrightarrow{\gamma}_F [\Sigma']\, \varphi_1'}{[\Sigma]\, \varphi_1;\, \varphi_2 \oplus \varphi_3 \xrightarrow{\gamma}_F [\Sigma']\, \varphi_1';\, \varphi_2 \oplus \varphi_3}$$

By this reduction rule we know that:

$$[\Lambda(\sigma)]\, \varphi_1 \xrightarrow{\gamma} [\Lambda(\sigma')]\, \varphi_1'$$

Hence we can use the inductive hypothesis to show that:

$$[\sigma]\, t_1 \xrightarrow{\gamma} [\sigma']\, t_1'$$

and hence that:

$$\frac{[\sigma']\, t_1 \xrightarrow{\gamma}_F [\sigma]\, t_1'}{[\sigma]\, \texttt{if}\, t_1 \,\texttt{then}\, t_2 \,\texttt{else}\, t_3 \xrightarrow{\gamma}_F [\sigma']\, \texttt{if}\, t_1' \,\texttt{then}\, t_2 \,\texttt{else}\, t_3}$$

Case: $t = \texttt{case}\,\alpha(\widetilde{T})\,\texttt{in}\,\{\widetilde{T \mapsto t}\}$

$$\frac{\varphi_i \wr \Gamma \vdash t_i \colon T \qquad \epsilon \wr \Gamma \vdash v_i \colon T_i \qquad T_i \neq \mathsf{Res}\,r}{\&(\alpha(\widetilde{T}), T_i);\ \varphi_i \wr \Gamma \vdash \texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \colon T}$$

By the hypothesis we know that

$$[\Sigma]\ \&(\alpha(\widetilde{T}), T);\ \varphi_i \to\ [\Sigma']\ \varphi'$$

The reduction rules for $\&(\alpha(\widetilde{T}), T);\ \varphi_i$ that fulfils these conditions are:

$$\frac{\exists i\,.\,([\Sigma]\,\varphi_i \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi'_i \,\wedge\, \nexists\texttt{error} \in \varphi'_i)}{[\Sigma]\,\varphi_1 \& \varphi_2 \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi'_i} \qquad \frac{[\Sigma]\,\varphi_2 \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi'_2 \quad \nexists\texttt{error} \in \varphi'_2}{[\Sigma]\,\varphi_1 \& \varphi_2 \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi'_2}$$

$$\frac{\forall i\,.\,([\Sigma]\,\varphi_i \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma_1]\,\varphi'_i \,\wedge\, \texttt{error} \in \varphi'_i)}{[\Sigma]\,\varphi_1 \& \varphi_2 \xrightarrow{\alpha(\widetilde{T})_j}_F [\Sigma_i]\,\varphi'_j}$$

Each of these can only reduce if:

$$[\Sigma]\ (\alpha(\widetilde{T}), T_i) \to$$

By our assumptions we know that $\Lambda(\sigma) \xrightarrow{\alpha(\widetilde{T})} \Lambda(\sigma') \Leftrightarrow \sigma \xrightarrow{\alpha(\widetilde{v})} \sigma'$.

Hence we can reduce $(\alpha(\widetilde{v}), T)$ using either the RCaseSucc or the RCaseErr rule:

$$\frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v})) \colon T_i \qquad T_i \in \widetilde{T}}{[\sigma]\,\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\,t_i} \qquad \frac{\sigma \xrightarrow{\alpha(\widetilde{v})} \sigma' \qquad \sigma(\alpha(\widetilde{v})) \colon T' \qquad T' \notin \widetilde{T}}{[\sigma]\,\texttt{case}\,\alpha(\widetilde{v})\,\texttt{in}\,\{\widetilde{T \mapsto t}\} \xrightarrow{\alpha(\widetilde{v})} [\sigma']\,\texttt{error}}$$

Case: $t = f(t_1)$

$$\frac{\varphi_1 \wr \Gamma \vdash t_1 \colon \textsc{Unit}}{f(\varphi_1) \wr \Gamma \vdash f(t_1) \colon \textsc{Unit}}$$

By the hypothesis we know that:

$$[\Sigma]\ f(\varphi_1) \to\ [\Sigma']\ \varphi'$$

There are three possible reduction rules for $f(\varphi_1)$.

Case RTRegOne

$$\frac{[\Sigma]\,\varphi_1 \xrightarrow{\tau}_F [\Sigma']\,\varphi'}{[\Sigma]\,f(\varphi_1) \xrightarrow{\tau}_F [\Sigma']\,f(\varphi'_1)}$$

As

$$[\Lambda(\sigma)]\,\varphi_1 \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\varphi_1'$$

we can apply the inductive hypothesis and show that:

$$[\sigma]\,t_1 \xrightarrow{\gamma} [\sigma']\,t_1'$$

We can then show that:

$$\frac{[\sigma]\,t_1 \xrightarrow{\tau}_F [\sigma']\,t_1'}{[\sigma']\,f(t_1) \xrightarrow{\tau}_F [\sigma']\,f(t_1')}$$

Case RTRegTwo

$$\frac{[\Sigma]\,\varphi_1 \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi'}{[\Sigma]\,f(\varphi_1) \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\,\varphi_1'}$$

As

$$[\Lambda(\sigma)]\,\varphi_1 \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\varphi_1'$$

we can apply the inductive hypothesis and show that:

$$[\sigma]\,t_1 \xrightarrow{\gamma} [\sigma']\,t_1'$$

We can then show that:

$$\frac{[\sigma]\,t_1 \xrightarrow{\alpha(\widetilde{v})}_F [\sigma']\,t_1'}{[\sigma]\,f(t_1) \xrightarrow{\alpha(\widetilde{v})}_{F\cup\{f\}} [\sigma']\,t_1'}$$

Case RTRegThree

$$\frac{}{[\Sigma]\,f(\epsilon) \xrightarrow{\tau} [\Sigma]\,\epsilon}$$

When exp $= v$ We can straightforwardly use the following reduction rule:

$$\frac{}{[\sigma]\,f(v) \xrightarrow{\tau} [\sigma]\,v}$$

Otherwise we can use RTRegOne or RTRegTwo.

Case: $t = \texttt{dmod}_{F_1}(t_1)$

$$\frac{\varphi_1 \wr \emptyset \vdash t_1 \colon \textsc{Unit}}{\texttt{dmod}_{\gamma_1}(\varphi_1) \wr \emptyset \vdash \texttt{dmod}_{\gamma_1}(t_1) \colon \textsc{Unit}}$$

There are two possible reductions for $\texttt{dmod}_{F_1}(\varphi_1)$. We can perform a case split on the rules' premises:

Case $\texttt{regions}(\varphi) \not\subseteq \gamma_1$

$$\frac{[\Sigma]\,\varphi_1 \xrightarrow{\gamma}_{F'} [\Sigma']\,\varphi_1' \qquad \texttt{regions}(\varphi_1) \not\subseteq F}{[\Sigma]\,\texttt{dmod}_F(\varphi_1) \xrightarrow{\gamma}_{F'} [\Sigma']\,\texttt{dmod}_{F \cup F'}(\varphi_1')}$$

As

$$[\Lambda(\sigma)]\,\varphi_1 \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\varphi_1'$$

we can apply the inductive hypothesis and show that:

$$[\sigma]\,t \xrightarrow{\gamma} [\sigma']\,t'$$

and hence can show that:

$$\frac{[\sigma]\,t \xrightarrow{\gamma}_{F'} [\sigma']\,t' \qquad \texttt{regions}(t) \not\subseteq F}{[\sigma]\,\texttt{dmod}_F(t) \xrightarrow{\gamma}_{F'} [\sigma']\,\texttt{dmod}_{F \cup F'}(t')}$$

Case $\texttt{regions}(\varphi) \subseteq \gamma_1$

$$\frac{\texttt{regions}(\varphi_1) \subseteq F}{[\Sigma]\,\texttt{dmod}_F(\varphi_1) \xrightarrow{\tau} [\Sigma]\,\varphi_1}$$

In this case we can straightforwardly apply the following reduction rule:

$$\frac{\texttt{regions}(t) \subseteq F}{[\sigma]\,\texttt{dmod}_F(t) \xrightarrow{\tau} [\sigma]\,t}$$

$\square$

A.15 Empty Effect Thread Liveness Lemma

Lemma 3.5 If:

$$\vdash P : \Phi \qquad \Phi = \Pi_I\, \varphi_i \qquad (\varphi_i \equiv \epsilon \vee \varphi_i \equiv \underline{X})$$

then

$$[\sigma]\, P \xrightarrow{\gamma} [\sigma]\, P' \vee P = \Pi_I\, v$$

Proof. **Case** $\Phi = \varphi$

$$\frac{\varphi \wr \emptyset \vdash t : T}{\vdash t : \varphi}$$

By Lemma 3.3 we know that:

$$[\sigma]\, t \xrightarrow{\gamma} [\sigma]\, t' \vee t = v$$

Case: $\Phi = \Phi_1 \parallel \Phi_1$

$$\frac{\vdash P_1 : \Phi_1 \qquad \vdash P_2 : \Phi_2}{\vdash P_1 \parallel P_2 : \Phi_1 \parallel \Phi_2}$$

We can then use the inductive hypothesis to show that:

$$([\sigma]\, P_1 \xrightarrow{\gamma} [\sigma]\, P_1' \vee P_1 = \Pi\, v)$$

$$([\sigma]\, P_2 \xrightarrow{\gamma} [\sigma]\, P_2' \vee P_2 = \Pi\, v)$$

and hence that:

$$[\sigma]\, P_1 \parallel P_2 \xrightarrow{\gamma} [\sigma]\, P_1' \parallel P_2 \vee [\sigma]\, P_1 \parallel P_2 \xrightarrow{\gamma} [\sigma]\, P_1 \parallel P_2' \vee P_1 \parallel P_2 = \Pi\, v$$

□

A.16 Active Effect Thread Liveness Lemma

Lemma 3.6

If:

$$\vdash P \colon \Phi \qquad [\Lambda(\sigma)]\,\Phi \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\Phi'$$

then

$$[\sigma]\,P \xrightarrow{\gamma} [\sigma']\,P'$$

Proof. By induction over $[\Lambda(\sigma)]\,\Phi \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\Phi'$

Case **RTAccSucc, RTSeq, RTIntChoice, RTExtChoiceOne, RTExtChoiceTwo, RTExtChoiceThree, RTRegOne, RTRegTwo, RTRegThree, RTDmodOne, RTDmodTwo**

$$\frac{\varphi \wr \emptyset \vdash t \colon T}{\vdash t \colon \varphi}$$

By Lemma 3.4 we know that:

$$[\sigma]\,t \xrightarrow{\gamma} [\sigma']\,t'$$

Case: **RTParOne**

$$\frac{\vdash P_1 \colon \Phi_1 \qquad \vdash P_2 \colon \Phi_2}{\vdash P_1 \parallel P_2 \colon \Phi_1 \parallel \Phi_2} \qquad \frac{[\Sigma]\,\Phi_1 \xrightarrow{\gamma}_F [\Sigma']\,\Phi_1'}{[\Sigma]\,\Phi_1 \parallel \Phi_2 \xrightarrow{\gamma}_F [\Sigma']\,\Phi_1' \parallel \Phi_2}$$

By the reduction rule we know that:

$$[\Lambda(\sigma)]\,\Phi_1 \xrightarrow{\gamma'} [\Lambda(\sigma')]\,\Phi_1'$$

We can then use the inductive hypothesis to show that:

$$[\sigma]\,P_1 \xrightarrow{\gamma} [\sigma']\,P_1'$$

and hence that:

$$\frac{[\sigma]\,P_1 \xrightarrow{\gamma}_F [\sigma']\,P_1'}{[\sigma]\,P_1 \parallel P_2 \xrightarrow{\gamma}_F [\sigma']\,P_1' \parallel P_2}$$

Case: **RTParTwo**

Similar to case RTParOne.

Case: RTEquiv

$$\frac{\Phi \equiv \Phi_1 \quad [\Sigma]\,\Phi_1 \xrightarrow{\gamma} [\Sigma']\,\Phi_2 \quad \Phi_2 \equiv \Phi'}{[\Sigma]\,\Phi \xrightarrow{\gamma} [\Sigma']\,\Phi'}$$

By induction. □

Appendix B

Blocking Message Passing

In this chapter we provide the full proofs of how the blocking message passing approach in Section 3.5.1 implies General Compatibility of static programs.

B.1 Separate Channels Lemma

Lemma B.1.

$$[\Sigma]\, \varphi_i \to [\Sigma']\, \varphi_i' \Leftrightarrow [c \mapsto q;\, \Sigma]\, \varphi_i \to [c \mapsto q;\, \Sigma']\, \varphi_i'$$

where

$$c \neq dd_i \wedge \varphi_i \textit{ has principle channel allocation}$$

Proof. By induction over $\to$

Case: RTAccSucc

$$\frac{\Sigma \xrightarrow{\alpha(\widetilde{T})} \Sigma' \quad \Sigma(\alpha(\widetilde{T})) = T}{[\Sigma]\, (\alpha(\widetilde{T}), T) \xrightarrow{\alpha(\widetilde{T})} [\Sigma']\, \epsilon}$$

The only actions we can perform are sending and receiving.

Case $c!\langle T\rangle$

Here $\Sigma' = \Sigma[c \mapsto q', T]$. Hence:

$$[\Sigma[c \mapsto q']]\, c!\langle T\rangle \to [\Sigma[c \mapsto q', T]]\, \epsilon \Leftrightarrow [\Sigma[c \mapsto q, q']]\, c!\langle T\rangle \to [\Sigma[c \mapsto q, q', T]]\, \varphi_1'$$

Case $c'?(T)$

As $c \neq dd_i$ and φ_i has principle channel allocation we know that $c' \neq c$. Hence:

$[\Sigma[c' \mapsto q']]\, c?(T) \rightarrow [\Sigma[c' \mapsto q'', T]]\, \epsilon \Leftrightarrow [c \mapsto q;\, \Sigma[c' \mapsto q']]\, c?(T) \rightarrow [c \mapsto q;\, \Sigma[c' \mapsto q'']]\, \varphi_1'$

Case: RTAccErr

Similar to case RTAccSucc

Case: RTSeq

$$\frac{[\Sigma]\, \varphi_1 \xrightarrow{\gamma}_F [\Sigma']\, \varphi_1'}{[\Sigma]\, \varphi_1;\, \varphi_2 \xrightarrow{\gamma}_F [\Sigma']\, \varphi_1';\, \varphi_2}$$

By induction we know that:

$$[\Sigma]\, \varphi_1 \rightarrow [\Sigma']\, \varphi_1' \Leftrightarrow [c \mapsto q;\, \Sigma]\, \varphi_1 \rightarrow [c \mapsto q;\, \Sigma']\, \varphi_1'$$

Hence we have that:

$$[\Sigma]\, \varphi_1;\, \varphi_2 \rightarrow [\Sigma']\, \varphi_1';\, \varphi_2 \Leftrightarrow [c \mapsto q;\, \Sigma]\, \varphi_1;\, \varphi_2 \rightarrow [c \mapsto q;\, \Sigma']\, \varphi_1';\, \varphi_2$$

Case: RTIntChoice

$$\frac{i \in 1, 2}{[\Sigma]\, \varphi_1 \oplus \varphi_2 \xrightarrow{\tau} [\Sigma]\, \varphi_i}$$

Trivial

Case: RTExtChoiceOne

$$\frac{\exists i\,.\,([\Sigma]\, \varphi_i \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\, \varphi_i' \wedge \nexists \texttt{error} \in \varphi_i')}{[\Sigma]\, \varphi_1 \& \varphi_2 \xrightarrow{\alpha(\widetilde{T})}_F [\Sigma']\, \varphi_i'}$$

By induction we know that:

$$[\Sigma]\, \varphi_i \rightarrow [\Sigma']\, \varphi_i' \Leftrightarrow [c \mapsto q;\, \Sigma]\, \varphi_i \rightarrow [c \mapsto q;\, \Sigma']\, \varphi_i'$$

Hence we have that:

$$[\Sigma]\, \varphi_1 \& \varphi_2 \rightarrow [\Sigma']\, \varphi_i' \Leftrightarrow [c \mapsto q;\, \Sigma]\, \varphi_1 \& \varphi_2 \rightarrow [c \mapsto q;\, \Sigma']\, \varphi_i'$$

Case: RTExtChoiceTwo

Similar to case RTExtChoiceOne.

Case: RTUnfold

$$\frac{}{[\Sigma]\,\mu\underline{X}.\varphi \xrightarrow{\tau} [\Sigma]\,\varphi[\mu\underline{X}.\varphi/\underline{X}]}$$

Trivial

Case: RTEquiv

$$\frac{\varphi \equiv \varphi_1 \quad [\Sigma]\,\varphi_1 \xrightarrow{\gamma} [\Sigma']\,\varphi_2 \quad \varphi_2 \equiv \varphi'}{[\Sigma]\,\varphi \xrightarrow{\gamma} [\Sigma']\,\varphi'}$$

By induction we know that:

$$[\Sigma]\,\varphi_1 \to [\Sigma']\,\varphi_2 \Leftrightarrow [c \mapsto q;\, \Sigma]\,\varphi_1 \to [c \mapsto q;\, \Sigma']\,\varphi_2$$

Hence we have that:

$$[\Sigma]\,\varphi \to [\Sigma']\,\varphi' \Leftrightarrow [c \mapsto q;\, \Sigma]\,\varphi \to [c \mapsto q;\, \Sigma']\,\varphi'$$

$\square$

B.2 Send Action Equivalence Lemma

Lemma B.2. *If*

$$\Pi\, G \restriction d_i \equiv \Pi\, \varphi_i \;\wedge\; \varphi_a \equiv \oplus_N(c!\langle T_n\rangle;\, \varphi_b^n) \;\wedge\; N \neq \emptyset$$

Then

$$\exists G' = d_a \to d_b \colon c\langle\widetilde{T \mapsto G}\rangle_M \,.\, N \subseteq M \;\wedge\; \Pi\, G' \restriction d_i \equiv \Pi\, \varphi_i$$

Proof. By induction over G

Case $G = \mathsf{0}, X$

$$G \restriction d_a \not\equiv \oplus_N(c!\langle T_n\rangle;\, \varphi_b^n)$$

Hence we can disregard these cases.

Case $G = \mu X.G''$

By equivalence we know that:

$$\begin{array}{rcl} \mu\underline{X}.G'' \restriction d_a & \equiv & \mu\underline{X}.\oplus_N (c!\langle T_n\rangle;\, \varphi_b^n) \\ & \equiv & \oplus_N(c!\langle T_n\rangle;\, \varphi_b^n)[\mu\underline{X}.\oplus_N (c!\langle T_n\rangle;\, \varphi_b^n)/\underline{X}] \\ & \equiv & (G''[\mu\underline{X}.G''/\underline{X}]) \restriction d_a \end{array}$$

Hence we can consider $(G'[\mu\underline{X}.G'/\underline{X}]) \restriction d_a$ directly.

Case $G = d_{a'} \to d_{b'} \colon c'\langle \widetilde{T \mapsto G} \rangle_M$

$c' = c$

We let $G' = G$ directly.

$c' \neq c$

Hence $a' \neq a$, $b' \neq b$. By the definition of $\upharpoonright$ we know that:

$$G_m \upharpoonright d_a \equiv \oplus_N (c!\langle T_n \rangle; \varphi_b^n)$$

Let $\Phi_m \equiv \Pi\, G_m \upharpoonright d_i$. Then, by induction we can show that:

$$\exists G'_m \,.\, \Phi_m \equiv \Pi\, G'_m \upharpoonright d_i \qquad G'_m = d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_O \qquad N \subseteq O$$

Then let:

$$G' = d_a \to d_b \colon c\langle \widetilde{T_o \mapsto d_{a'} \to d_{b'} \colon c'\langle \widetilde{T_m \mapsto G_o} \rangle_M} \rangle_O$$

Here:

$$\Pi\, G' \upharpoonright d_i \equiv \Pi\, \varphi_i$$

$\square$

B.3 Receive Action Equivalence Lemma

Lemma B.3. *If*

$$\Pi\, G \restriction d_i \equiv \Pi\, \varphi_i \;\wedge\; \varphi_b \equiv \&_M (c!\langle T_m \rangle;\, \varphi_b^m)$$

Then

$$\exists G' = d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_N \,.\, \Pi\, G' \restriction d_i \equiv \Pi\, \varphi_i$$

Proof. Similar to Lemma B.2. □

B.4 Valid States Safety Lemma

Lemma 3.10

$$[\Sigma]\,\Phi \in \Delta \Rightarrow \texttt{error} \notin \Phi$$

Proof. By induction over n.

Case $n = 0$

$$\Delta_0 \overset{\text{def}}{=} \{[\Sigma_\emptyset]\,\Phi \mid \Phi \equiv \Pi\, G \restriction d_i\}$$

By induction over G

Case $G = \mathtt{0}, X$

Trivial.

Case $G = \mu X.G$

By induction we have that:

$$\Phi' = \Pi\,\varphi_i \equiv \Pi\, G \restriction d_i \Rightarrow \texttt{error} \notin \varphi$$

Hence:

$$\Phi = \Pi\,\mu\underline{X}.\varphi_i \equiv \Pi\,\mu\underline{X}.G \restriction d_i \Rightarrow \texttt{error} \notin \varphi'$$

Case $G = d_a \to d_b\colon c\langle\widetilde{T \mapsto G}\rangle$

$$\begin{array}{lcll} d_1 \to d_2\colon r\langle\widetilde{T \mapsto G}\rangle_M \restriction d_1 & \overset{\text{def}}{=} & \oplus_N(c!\langle T_n\rangle;\, G_n \restriction d_1) & \emptyset \neq N \subseteq M \\ d_1 \to d_2\colon r\langle\widetilde{T \mapsto G}\rangle \restriction d_2 & \overset{\text{def}}{=} & \&_M(c?(T_m);\, G_m \restriction d_1) & \\ d_1 \to d_2\colon r\langle\widetilde{T \mapsto G}\rangle \restriction d & \overset{\text{def}}{=} & G_1 \restriction d & \end{array}$$

By induction we know that:

$$\Phi'\Pi\,\varphi_i' \equiv \Pi\, G_m \restriction d_i \Rightarrow \texttt{error} \notin \Phi'$$

Hence:

$$\varphi_i^a \equiv \oplus_N(c!\langle T_n\rangle;\, G_n \restriction d_1) \Rightarrow \texttt{error} \notin \varphi_i^a$$

$$\varphi_i^n \equiv \&_M(c?(T_m);\, G_m \restriction d_1) \Rightarrow \texttt{error} \notin \varphi_i^b$$

Therefore we have that:

$$\Phi \equiv \Pi\, d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle \restriction d_i \Rightarrow \texttt{error} \notin \Phi$$

Case $n > 0$

$\forall \varphi_i, c = d_a d_b \,\forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \varphi_m \qquad k \in J$$

$\forall \varphi_i, c = d_a d_b \,\forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \parallel \varphi_a^k \parallel \ldots \varphi_m \qquad k \in J$$

By induction we know that:

$$[\Sigma]\, \Phi' = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n \Rightarrow \texttt{error} \notin \Phi'$$

Hence we have that:

$$\texttt{error} \notin \&_J(c?(T_j);\, \varphi_b^j)$$

Therefore we can show that:

$$[\_]\, \Phi = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \varphi_m \Rightarrow \texttt{error} \notin \Phi$$

□

B.5 Valid States Liveness Lemma

Lemma 3.11
$$[\Sigma]\,\Phi \in \Delta \Rightarrow [\Sigma]\,\Phi \rightarrow [\Sigma']\,\Phi' \;\vee\; \Phi \equiv \Pi\,\epsilon \;\vee\; \Phi \equiv \Pi\,\underline{X}$$

Proof. By induction over n.

Case $n = 0$

By case analysis of G.

If $[\Sigma_\emptyset]\,\Phi \in \Delta_0$ and $\Phi \equiv \Phi'$ then $[\Sigma_\emptyset]\,\Phi' \in \Delta_0$ and hence it is sufficient to consider direct reduction.

Case $G = d_a \rightarrow d_b\colon c\langle \widetilde{T \mapsto G}\rangle$

If
$$d_a \rightarrow d_b\colon c\langle \widetilde{T \mapsto G}\rangle \restriction d_a \equiv \oplus_N(c!\langle T_n\rangle;\, G_n \restriction d_a)$$
then:

$$[\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel \oplus_N(c!\langle T_n\rangle;\, G_n \restriction d_a) \parallel \ldots \parallel \varphi_o \rightarrow [\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel (c!\langle T_n\rangle;\, G_n \restriction d_a) \parallel \ldots \parallel \varphi_o$$

If
$$d_a \rightarrow d_b\colon c\langle \widetilde{T \mapsto G}\rangle \restriction d_a \equiv \varphi_1 \parallel \ldots \parallel (c!\langle T_n\rangle;\, G_n \restriction d_a) \parallel \ldots \parallel \varphi_o$$
then:

$$[\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel (c!\langle T_n\rangle;\, G_n \restriction d_a) \parallel \ldots \parallel \varphi_o \rightarrow [c \mapsto T_n]\,\varphi_1 \parallel \ldots \parallel G_n \restriction d_a \parallel \ldots \parallel \varphi_o$$

Case $G = 0$

$$\Phi \equiv \Pi\,\epsilon$$

Case $G = X$

$$\Phi \equiv \Pi\,\underline{X}$$

Case $n > 0$

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \varphi_m \qquad k \in J$$

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \parallel \varphi_a^k \parallel \ldots \varphi_m \qquad k \in J$$

We can show that:

$$[c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \varphi_m \to [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m$$

$\square$

Appendix C

Non Blocking Message Passing

In this chapter we provide the full proofs of how the non-blocking message passing approach in Section 3.5.2 implies General Compatibility of static programs.

C.1 Non Blocking Send Action Equivalence Lemma

Lemma C.1. *If*

$$\varphi_a \equiv \oplus_N((c!\langle T_n\rangle, \textsc{Unit});\, G_n \upharpoonright d_1 \& (c!\langle T_n\rangle, l_{[]});\, G_{l_{[]}} \upharpoonright d_1)$$

$$\Pi\, G \upharpoonright d_i \equiv \Pi\, \varphi_i \qquad N \neq \emptyset$$

Then

$$\exists G' = d_a \to d_b \colon c\langle \widetilde{T \mapsto G}\rangle_M \,.\, N \subseteq M \qquad \Pi\, G' \upharpoonright d_i \equiv \Pi\, \varphi_i$$

Proof. By induction over G

Case $G = \mathsf{0}, X$

$$G \upharpoonright d_a \not\equiv \oplus_N(c!\langle T_n\rangle;\, \varphi_b^n)$$

Hence we can disregard these cases.

Case $G = \mu X.G''$

By equivalence we know that:

$$\begin{array}{rcl}\mu\underline{X}.G'' \upharpoonright d_a & \equiv & \mu\underline{X}. \oplus_N (c!\langle T_n\rangle;\, \varphi_b^n) \\ & \equiv & \oplus_N(c!\langle T_n\rangle;\, \varphi_b^n)[\mu\underline{X}. \oplus_N (c!\langle T_n\rangle;\, \varphi_b^n)/\underline{X}] \\ & \equiv & (G''[\mu\underline{X}.G''/\underline{X}]) \upharpoonright d_a\end{array}$$

Hence we can consider $(G'[\mu\underline{X}.G'/\underline{X}]) \upharpoonright d_a$ directly.

Case $G = d_{a'} \to d_{b'}\colon c'\langle\widetilde{T \mapsto G}\rangle_M$

$c' = c$

We let $G' = G$ directly.

$c' \neq c$

Hence $a' \neq a$, $b' \neq b$. By the definition of $\upharpoonright$ we know that:

$$G_m \upharpoonright d_a \equiv \oplus_N((c!\langle T_n\rangle, \text{UNIT});\, G_n \upharpoonright d_1 \& (c!\langle T_n\rangle, l_{[]});\, G_{l_{[]}} \upharpoonright d_1)$$

Let $\Phi_m \equiv \Pi\, G_m \upharpoonright d_i$. Then, by induction we can show that:

$$\exists G'_m \,.\, \Phi_m \equiv \Pi\, G'_m \upharpoonright d_i \qquad G'_m = d_a \to d_b\colon c\langle\widetilde{T \mapsto G}\rangle_O \qquad N \subseteq O$$

Then let:

$$G' = d_a \to d_b\colon c\langle T_o \mapsto \widetilde{d_{a'} \to d_{b'}}\colon c'\langle\widetilde{T_m \mapsto G_o}\rangle_M\rangle_O$$

Here:

$$\Pi\, G' \upharpoonright d_i \equiv \Pi\, \varphi_i$$

□

C.2 Non Blocking Receive Action Equivalence Lemma

Lemma C.2. *If*

$$\Pi\, G \upharpoonright d_i \equiv \Pi\, \varphi_i \;\wedge\; \varphi_b \equiv \&_M (c!\langle T_m \rangle;\, \varphi_b^m)$$

Then

$$\exists G' = d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle_N \,.\, \Pi\, G' \upharpoonright d_i \equiv \Pi\, \varphi_i$$

Proof. Similar to Lemma C.1. □

C.3 Valid States Safety Lemma

Lemma 3.16

$$[\Sigma]\,\Phi \in \Delta \Rightarrow \texttt{error} \notin \Phi$$

Proof. By induction over n.

Case $n = 0$

$$\Delta_0 \overset{\text{def}}{=} \{[\Sigma_\emptyset]\,\Phi \mid \Phi \equiv \Pi\, G \upharpoonright d_i\}$$

By induction over G

Case $G = \mathtt{0}, X$

Trivial.

Case $G = \mu X.G$

By induction we have that:

$$\Phi' = \Pi\,\varphi_i \equiv \Pi\, G \upharpoonright d_i \Rightarrow \texttt{error} \notin \varphi$$

Hence:

$$\Phi = \Pi\,\mu\underline{X}.\varphi_i \equiv \Pi\,\mu\underline{X}.G \upharpoonright d_i \Rightarrow \texttt{error} \notin \varphi'$$

Case $G = d_a \to d_b\colon c\langle \widetilde{T \mapsto G}\rangle$

$$\begin{array}{lcl}
d_1 \to d_2\colon r\langle \widetilde{T \mapsto G}\rangle_M \upharpoonright d_1 & \overset{\text{def}}{=} & \oplus_N((c!\langle T_n\rangle, \textsc{Unit});\, G_n \upharpoonright d_1 \& (c!\langle T_n\rangle, l_\emptyset);\, G_{l_\emptyset} \upharpoonright d_1) \\
 & & \quad \emptyset \neq N \cup l_\emptyset \mapsto G_{l_\emptyset} \subseteq M \\
d_1 \to d_2\colon r\langle \widetilde{T \mapsto G}\rangle \upharpoonright d_2 & \overset{\text{def}}{=} & \&_M(c?(T_m);\, G_m \upharpoonright d_1) \\
d_1 \to d_2\colon r\langle \widetilde{T \mapsto G}\rangle \upharpoonright d & \overset{\text{def}}{=} & G_1 \upharpoonright d
\end{array}$$

By induction we know that:

$$\Phi'\Pi\,\varphi'_i \equiv \Pi\, G_m \upharpoonright d_i \Rightarrow \texttt{error} \notin \Phi'$$

Hence:

$$\varphi_i^a \equiv \oplus_N((c!\langle T_n\rangle, \textsc{Unit});\, G_n \upharpoonright d_1 \& (c!\langle T_n\rangle, l_\emptyset);\, G_{l_\emptyset} \upharpoonright d_1) \Rightarrow \texttt{error} \notin \varphi_i^a$$

$$\varphi_i^n \equiv \&_M(c?(T_m);\, G_m \upharpoonright d_1) \Rightarrow \texttt{error} \notin \varphi_i^b$$

Therefore we have that:

$$\Phi \equiv \Pi\, d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle \upharpoonright d_i \Rightarrow \texttt{error} \notin \Phi$$

Case $n > 0$

Case: type in queue

$\forall \varphi_i, c = d_a d_b\, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \varphi_m \qquad k \in J$$

$\forall \varphi_i, c = d_a d_b\, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \parallel \varphi_a^k \parallel \ldots \varphi_m \qquad k \in J$$

By induction we know that:

$$[\Sigma]\, \Phi' = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n \Rightarrow \texttt{error} \notin \Phi'$$

Hence we have that:

$$\texttt{error} \notin \&_J(c?(T_j);\, \varphi_b^j)$$

Therefore we can show that:

$$[\_]\, \Phi = [c \mapsto T_k;\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j);\, \varphi_b^j) \parallel \ldots \varphi_m \Rightarrow \texttt{error} \notin \Phi$$

Case: type hole in queue

$\forall \varphi_i, c = d_a d_b\, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\Sigma[c \mapsto q_{[]}] \qquad \delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto [];\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^{l_{[]}} \parallel \ldots \parallel$$
$$\oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_m \qquad k \in J$$

$\forall \varphi_i, c = d_a d_b\, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\Sigma[c \mapsto q_{[]}] \qquad \delta_j = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto [];\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel$$
$$\varphi_b^{l_{[]}} \parallel \ldots \parallel \varphi_m \qquad k \in J$$

By induction we know that:

$$[\Sigma]\, \Phi' = [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n \Rightarrow \texttt{error} \notin \Phi'$$

Hence we have that:

$$\texttt{error} \notin \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}})$$

Therefore we can show that:

$$[\_]\, \Phi = [c \mapsto [];\, \Sigma]\, \varphi_1 \parallel \ldots \parallel \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_b^{l_{[]}} \parallel \ldots \varphi_m$$

$$\Rightarrow \texttt{error} \notin \Phi$$

$\square$

C.4 Valid States Liveness Lemma

Lemma 3.17

$$[\Sigma]\,\Phi \in \Delta \Rightarrow [\Sigma]\,\Phi \to [\Sigma']\,\Phi' \;\vee\; \Phi \equiv \Pi\,\epsilon \;\vee\; \Phi \equiv \Pi\,\underline{X}$$

Proof. By induction over n.

Case $n = 0$

By case analysis of G.

If $[\Sigma_\emptyset]\,\Phi \in \Delta_0$ and $\Phi \equiv \Phi'$ then $[\Sigma_\emptyset]\,\Phi' \in \Delta_0$ and hence it is sufficient to consider direct reduction.

Case $G = d_a \to d_b\colon c\langle \widetilde{T \mapsto G} \rangle$

If

$$d_a \to d_b\colon c\langle \widetilde{T \mapsto G}\rangle \restriction d_a \equiv \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}})$$

then:

$$\begin{aligned}&[\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_o \to\\ &\quad [\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel ((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_o\end{aligned}$$

If

$$d_a \to d_b\colon c\langle \widetilde{T \mapsto G}\rangle \restriction d_a \equiv ((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}})$$

then:

$$\begin{aligned}&[\Sigma_\emptyset]\,\varphi_1 \parallel \ldots \parallel ((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k)\&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_o \to\\ &\quad [c \mapsto T_n]\,\varphi_1 \parallel \ldots \parallel G_n \restriction d_a \parallel \ldots \parallel \varphi_o\end{aligned}$$

Case $G = \mathsf{0}$

$$\Phi \equiv \Pi\,\epsilon$$

Case $G = X$

$$\Phi \equiv \Pi\,\underline{X}$$

Case $n > 0$

Case: type in queue

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \varphi_m \qquad k \in J$$

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\delta'_k = [c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \parallel \varphi_a^k \parallel \ldots \varphi_m \qquad k \in J$$

We can show that:

$$[c \mapsto T_k; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \&_J(c?(T_j); \, \varphi_b^j) \parallel \ldots \varphi_m \to [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^k \parallel \ldots \parallel \varphi_b^j \parallel \ldots \varphi_m$$

Case: type hole in queue

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\Sigma[c \mapsto q_{[]}] \qquad \delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\begin{aligned} \delta'_k = [c \mapsto []; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_b^{l_{[]}} \parallel \ldots \parallel \\ \oplus_N((c!\langle T_k \rangle, \textsc{Unit}); \, \varphi_a^k) \& ((c!\langle T_k \rangle, l_{[]}); \, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_m \qquad k \in J \end{aligned}$$

$\forall \varphi_i, c = d_a d_b \, \forall j \in J$ choose some $\varphi_a^j, \varphi_b, \Sigma$ such that:

$$\Sigma[c \mapsto q_{[]}] \qquad \delta_j = [\Sigma] \, \varphi_1 \parallel \ldots \parallel \varphi_a^j \parallel \ldots \parallel \varphi_b^j \parallel \ldots \parallel \varphi_m \qquad \delta_j \in \Delta_n$$

then $\delta'_k \in \Delta_{n+1}$ where:

$$\begin{aligned} \delta'_k = [c \mapsto []; \, \Sigma] \, \varphi_1 \parallel \ldots \parallel \oplus_N((c!\langle T_k \rangle, \textsc{Unit}); \, \varphi_a^k) \& ((c!\langle T_k \rangle, l_{[]}); \, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \\ \varphi_b^{l_{[]}} \parallel \ldots \parallel \varphi_m \qquad k \in J \end{aligned}$$

We can show that:

$$\begin{aligned}&[c \mapsto []; \Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^{l_{[]}} \parallel \ldots \parallel \oplus_N((c!\langle T_k\rangle, \textsc{Unit});\, \varphi_a^k) \&((c!\langle T_k\rangle, l_{[]});\, \varphi_a^{l_{[]}}) \parallel \ldots \parallel \varphi_m \\ &\qquad \rightarrow [\Sigma]\, \varphi_1 \parallel \ldots \parallel \varphi_b^{l_{[]}} \parallel \ldots \parallel \varphi_a^{l_{[]}} \parallel \ldots \parallel \varphi_m\end{aligned}$$

□

Appendix D

Global Typability and Local Update

In this chapter we provide the auxiliary proofs used to show that Global Typability and Local Update imply General Update Compatibility of dynamic programs.

D.1 Participants Relation as Strict Partial Order

Lemma D.1. *$<_G$ is a partial order.*

Proof. A partial order is anti-symmetric and transitive. We have transitivity straightforwardly from the rules. We then need to prove anti-symmetry: that $d_1 <_G d_2$ implies that $d_2 <_G d_1$ is not the case. We proceed by induction on the last derivation rule used for $d_1 <_G d_2$.

Consider the case where the rule used is that for $f(G')$, $\mu \underline{X}.G'$, or $\mathtt{dmod}_F(G')$. The conclusions then follow directly by application of the induction hypothesis.

Consider the case where the last rule used is:

$$\frac{d_1 <_{G_k} d_2 \quad d_2 \neq d, d'}{d_1 <_{d \to d' : c\langle \widetilde{T \mapsto G} \rangle} d_2}$$

By induction we know that it is not the case that $d_2 <_{G_k} d_1$, for some $G_k \in \widetilde{G}$. Suppose for contradiction that $d_2 <_G d_1$ also holds. In that case, we also have $d_2 <_{G'_k} d_1$ for some $G'_k \in \widetilde{G}$. We know that $G_k \upharpoonright d_2 = G_{k'} \upharpoonright d_2$ and by Lemma D.2 we also have $d_2 <_{G_k} d_1$. This contradicts the fact that $d_2 <_{G_k} d_1$ does not hold.

Consider the case where the last rule used is:

$$\frac{}{d_1 <_{d_1 \to d_2 : c\langle \widetilde{T \mapsto G} \rangle} d_2}$$

as the only possible rule that could derive $d_2 <_G d_1$ is an application of

$$\frac{d_1 <_{G_k} d_2 \quad d_2 \neq d, d'}{d_1 <_{d \to d' : c\langle \widetilde{T \mapsto G} \rangle} d_2}$$

on some $d' <_{G_k} d_1$ (to allow for transitivity) for some $G_k \in \widetilde{G}$. The side-condition $d_1 \neq d, d'$ cannot, however, hold in this case, so this is not possible. □

D.2 Equivalent Projection Give Equivalent Orderings

Lemma D.2. *If $d_1 <_G d_2$ and $G \restriction d_2 \equiv G' \restriction d_2$ then $d_1 <_{G'} d_2$*

Proof. By lexicographic induction on the structure of G then the structure of G'.

Consider the cases where G is 0 or $\underline{X}$. There are no derivation rule for $d_1 <_G d_2$ for such G, and hence we can disregard these cases.

Consider the cases where G is $\mu\underline{X}.G_1$, $f(G_1)$, $\mathtt{dmod}_F(G_1)$. Each case follows directly by induction.

Consider the case where G is $d \to d' : c\langle \widetilde{T \mapsto G} \rangle$. We proceed by a case split on the last rule used to derive $d_1 <_G d_2$.

If the last rule used is

$$\frac{}{d_1 <_{d_1 \to d_2 : c\langle \widetilde{T \mapsto G} \rangle} d_2}$$

Recall that here $d = d_1$ and $d' = d_2$. We then proceed by a case split over the structure of G'. In the case where $G' = \mu\underline{X}.G, 0$ or $\underline{X}$, then we have a contradiction, as $G' \restriction d_2 \not\equiv G \restriction d_2$. Consider the case where $G' = d_3 \to d_4 : c'\langle \widetilde{T' \mapsto G'} \rangle$. If we have that $d_4 = d_2$, then by the hypotheses we have that $G' \restriction d_2 = G \restriction d_2$. By principle channel allocation, and the fact that $c = d_1, d_2$ we also know that $d_1 = d_3$. Then by

$$\frac{}{d_1 <_{d_1 \to d_2 : c\langle \widetilde{T' \mapsto G'} \rangle} d_2}$$

we have that $d_1 <_G' d_2$. If we have that $d_4 \neq d_2$, then by the definition of projection we know that $G \restriction d_2 = G'_k \restriction d_2$, for some, for some $G_k \in \widetilde{G}$ and for some $G'_k \in \widetilde{G'}$. Then, by lexicographic induction we then can show that $d_1 <_{G'_k} d_2$, and hence that $d_1 <_{G'} d_2$.

If the last rule used is

$$\frac{d_1 <_{G_k} d_2 \quad d_2 \neq d, d'}{d_1 <_{d \to d' : c\langle \widetilde{T \mapsto G} \rangle} d_2}$$

Here we know that $d_2 \neq d, d'$. By the definition of projection we know that $G \restriction d_2 = G_k \restriction d_2$, for some $G_k \in \widetilde{G}$. By applying the inductive hypothesis we know that $d_1 <_{G'} d2$. as required.

□

D.3 Global Session Type Updatability Lemma

Lemma 5.4

If

$$\Phi \equiv \Pi_I\, G \upharpoonright d_i \qquad u = \widetilde{f_i \mapsto t_i}_I \qquad \varphi_i \wr \emptyset \vdash t_i \colon \textsc{Unit} \qquad G' \upharpoonright d_i \equiv \varphi_i$$

Then

$$\mathsf{upd}(\Phi, u, b) \equiv \Pi_I\, G'' \upharpoonright d_i$$

Proof. By induction over G.

Case $G = \mathsf{0}$

By the definition of $\upharpoonright$ we have that:

$$\mathsf{0} \upharpoonright d = \epsilon$$

Hence we have that:

$$\Phi \equiv \Pi_I\, \epsilon$$

By Lemma A.4 and the definition of `upd` we know that:

$$\mathsf{upd}(\Phi, u, b) \equiv \Pi_I\, \epsilon$$

Hence we can say that:

$$\mathsf{upd}(\Phi, u, b) \equiv \Pi_I\, \mathsf{0} \upharpoonright d_i$$

Case $G = X$

Similar to case: $G = \mathsf{0}$.

Case $G = d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle$

By the definition of $\upharpoonright$ we know that:

$$\Pi_I\, d_a \to d_b \colon c\langle \widetilde{T \mapsto G} \rangle \upharpoonright d_i \Pi\, \varphi_i$$

where:

$$\varphi_a \equiv \oplus(\mathtt{snd}(\mathsf{Res}\, r, T_1), \textsc{Unit});\ G_j \upharpoonright d_1 \qquad \varphi_b \equiv \&(\mathtt{rcv}(\mathsf{Res}\, r), T_i);\ G_j \upharpoonright d_2$$

By Lemma A.4 and the definition of `upd` we know that:

$$\mathsf{upd}((\alpha(\widetilde{v}), T);\, \varphi, u, b) \equiv (\alpha(\widetilde{v}), T);\, \mathsf{upd}(\varphi, u, b)$$

By the well formedness of Global Session Types we know that:

$$G \upharpoonright d_i \equiv G_j \upharpoonright d_i$$

where $i \in I \setminus a, b$. Using the inductive hypothesis we have that:

$$\Phi' \equiv \Pi_I\, G_j \upharpoonright d_i \Rightarrow \mathsf{upd}(\Phi', u, b) \equiv \Pi_I\, G'_j \upharpoonright d_i$$

Hence we have that:

$$\mathsf{upd}(\Pi_I\, d_a \to d_b \colon c\langle \widetilde{T \mapsto G_j} \rangle \upharpoonright d_i, u, b) \equiv \Pi_I\, d_a \to d_b \colon c\langle \widetilde{T \mapsto G'_j} \rangle \upharpoonright d_i$$

Case $G = \mu X.G_1$

By the definition we know that:

$$\mu \underline{X}.G_1 \upharpoonright d \equiv \mu \underline{X}.G_1 \upharpoonright d$$

By induction we know that:

$$\Pi\, \mathsf{upd}(G_1 \upharpoonright d, u, b) \equiv \Pi\, G''' \upharpoonright d$$

By Lemma A.4 and the definition of `upd` we know that:

$$\Pi\, \mathsf{upd}(\mu \underline{X}.G_1 \upharpoonright d, u, b) \equiv \Pi\, \mu \underline{X}.G''' \upharpoonright d$$

Case $G = f(G_1)$

Case $b = \texttt{true}$

By the definition of $\upharpoonright$ we know that:

$$\Pi\, f(G_1) \upharpoonright d \equiv \Pi\, f_d(G_1 \upharpoonright d)$$

By Lemma A.4 and the definition of `upd` we know that:

$$\begin{aligned} \Pi\, \mathsf{upd}(f(G_1) \upharpoonright d, u, \texttt{true}) &\equiv \Pi\, \mathsf{upd}(\varphi'_i, u, \texttt{true}) \\ &\equiv \Pi\, \mathsf{upd}(G' \upharpoonright d, u, \texttt{true}) \end{aligned}$$

Case $b = \texttt{false}$

By Lemma A.4 and the definition of `upd` we know that:

$$\Pi\, \mathsf{upd}(f(G_1) \restriction d, u, \texttt{false}) \quad \equiv \quad \Pi\, f_d(\mathsf{upd}(G_1 \restriction d, u, \texttt{false}))$$

By induction we know that:

$$\Pi\, \mathsf{upd}(G_1 \restriction d, u, \texttt{false}) \quad \equiv \quad \Pi\, G''' \restriction d$$

Hence we have that:

$$\Pi\, \mathsf{upd}(f(G_1) \restriction d, u, \texttt{false}) \quad \equiv \quad \Pi\, f_d(\mathsf{upd}(G''' \restriction d, u, \texttt{false}))$$

$$G'' = f(G''')$$

Case $G = \texttt{dmod}_F(G_1)$

By Lemma A.4 and the definition of `upd` we know that:

$$\Pi\, \mathsf{upd}(\texttt{dmod}_F(G_1) \restriction d, u, b) \quad \equiv \quad \Pi\, \texttt{dmod}_{F_d}(\mathsf{upd}(G_1 \restriction d, u, \texttt{true}))$$

where $\texttt{regions}(G_1) \cap F = \emptyset$ and

$$\Pi\, \mathsf{upd}(\texttt{dmod}_F(G_1) \restriction d, u, b) \quad \equiv \quad \Pi\, \texttt{dmod}_{F_d}(\mathsf{upd}(G_1 \restriction d, u, \texttt{false}))$$

where $\texttt{regions}(G_1) \cap F \neq \emptyset$. By induction we know that:

$$\Pi\, \mathsf{upd}(G_1 \restriction d, u, b) \quad \equiv \quad \Pi\, G''' \restriction d$$

Hence we have that:

$$\Pi\, \mathsf{upd}(\texttt{dmod}_F(G_1) \restriction d, u, b) \quad \equiv \quad \Pi\, (\texttt{dmod}_F(G''')) \restriction d$$

□

References

Ajmani, S. (2004). Automatic Software Upgrades for Distributed Systems. *Artificial Intelligence*, (Cm):1–23.

Ajmani, S., Liskov, B., and Shrira, L. (2006). Modular Software Upgrades for Distributed Systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 452–476. Massachusetts Institute of Technology.

Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. (2005). OPUS : Online Patches and Updates for Security. In *Proceedings of the 14th USENIX Security Symposium*, pages 287–302, Baltimore, Maryland, USA. USENIX Association.

Anderson, G. (2011). Dynamic Software Update for Behavioural Properties of Concurrent Programs. In *Proceedings of the 11th Grace Hopper Celebration of Women in Computing*, Portland, Oregon, USA.

Anderson, G. and Rathke, J. (2009). Migrating Protocols in Multi-threaded Message-passing Systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 8:1–8:5, Orlando, Florida, USA. ACM.

Anderson, G. and Rathke, J. (2011). Resource Access with Variably Typed Return. In *Proceedings of the 4th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, pages 51–57, Saarbrucken.

Anderson, G. and Rathke, J. (2012). Dynamic Software Update for Message Passing Programs. In *The Tenth International Asian Symposium on Programming Languages and Systems*, page (to appear), Kyoto, Japan. LNCS.

Appel, A. (1994). Hot-sliding in ML. Technical report, Princeton University, Princeton, New Jersey, USA.

Arthur, L. (1988). *Software Evolution: the Software Maintenance Challenge.* Wiley-Interscience, New York, New York, USA.

Banker, R., Datar, S., Kemerer, C., and Zweig, D. (1993). Software Complexity and Maintenance Costs. *Communications of the ACM*, 36(11):81–94.

Bartoletti, M., Degano, P., Ferrari, G., and Zunino, R. (2007). Types and effects for resource usage analysis. In *In: Proceedings of the 10th international conference on Foundations of software science and computational structures*, volume vol, pages 4423pp32–47.

Baumann, A., Appavoo, J., Wisniewski, R., Silva, D. D., Krieger, O., and Heiser, G. (2007). Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the 18th USENIX Annual Technical Conference*, pages 26:1–26:14, Santa Clara, California, USA. USENIX Association.

Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., Wisniewski, R. W., and Kerr, J. (2005). Providing dynamic update in an operating system. pages 32–32, Anaheim, CA. USENIX Association.

Bettini, L., Coppo, M., Dezani-Ciancaglini, M., Luca, M. D., and Yoshida, N. (2008). Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proceedings of the 19th International Conference on Concurrency Theory*, pages 418–433, Toronto, Ontario, Canada. Springer-Verlag.

Bierman, G., Hicks, M., Sewell, P., and Stoyle, G. (2003). Formalizing Dynamic Software Updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, Warsaw, Poland.

Bierman, G., Parkinson, M., and Noble, J. (2008). UpgradeJ : Incremental Typechecking for Class Upgrades. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 235–259, Paphos, Cypress. Springer-Verlag.

Boyapati, C., Liskov, B., Shrira, L., Moh, C.-h., and Richman, S. (2003). Lazy Modular Upgrades In Persistent Object Stores. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 403–417, Anaheim, California, USA. ACM.

Carbone, M., Honda, K., and Yoshida, N. (2007). Structured Communication-Centred Programming for Web Services. In *Proceedings of the 16th European Conference on Programming*, pages 2–17, Braga, Portugal. Springer-Verlag.

Cardelli, L. (1997). Program Fragments, Linking, and Modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France. ACM.

Cook, R. and Lee, I. (1983). DYMOS: a Dynamic Modification System. *SIGSOFT Software Engineering Notes*, 8(4):201–202.

Deniélou, P.-m. and Yoshida, N. (2010). Buffered Communication Analysis in Distributed Multiparty Sessions. In *Proceedings of the 21st International Conference on Concurrency Theory*, pages 343–357, Paris, France. Springer-Verlag.

Dezani-Ciancaglini, M. and Liguoro, U. D. (2010). Sessions and Session Types : an Overview. In *Proceedings of the 6th International Conference on Web Services and Formal Methods*, pages 1–28, Bologna, Italy. Springer-Verlag.

Dmitriev, M. (2001). Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, Tampa, Florida, USA.

Drossopoulou, S. and Eisenbach, S. (2002). Manifestations of Java Dynamic Linking - an Approximate Understanding at Source Language Level. In *Proceedings of the 1st International Workshop on Unanticipated Software Evolution*, Málaga, Spain. Springer-Verlag.

Duggan, D. (2005). Type-based Hot Swapping of Running Modules. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, volume 41, pages 181–220, Florence, Italy. ACM.

Eaddy, M. and Feiner, S. (2005). Multi-Language Edit-and-Continue for the Masses. Technical report, Columbia University, New York, New York, USA.

Erlang (2010). Appup Cookbook - Typical Cases of Upgrades/Downgrades Done in Runtime. http://www.erlang.org/doc/design_principles/appup_cookbook.html.

Flanagan, C. and Freund, S. (2000). Type-based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, volume 35, pages 219–232, Vancouver, British Columbia, Canada. ACM.

Freeman, P. (1986). Programming Without Tears. *High Technology*, 6(4):38–45.

Gay, S. and Hole, M. (2005). Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225.

Gay, S. and Vasconcelos, V. (2007). Asynchronous Functional Session Types. Technical report 2007–251, Department of Computing, University of Glasgow, Glasgow, Scotland, UK.

Gay, S. and Vasconcelos, V. (2010). Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50.

Gay, S., Vasconcelos, V., and Ravara, A. (2003). Session Types for Inter-Process Communication. Technical report, University of Glasgow, Glasgow, Scotland, UK.

Gifford, D. K. and Lucassen, J. M. (1986). Integrating Functional and Imperative Programming. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 28–38, Cambridge, Massachusetts, USA. ACM.

Gilmore, S., Kirli, D., and Walton, C. (1998). Dynamic ML without Dynamic Types. Technical report.

Gregersen, A. and Jorgensen, B. (2009). Dynamic Update of Java Applications - Balancing Change Fexibility vs Programming Transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112.

Gregersen, A. and Jorgensen, B. (2011). Run-time Phenomena in Dynamic Software Updating : Causes and Effects. In *Proceedings of the 12th International Workshop on Principles on Software Evolution*, pages 6–15, Szeged, Hungary. ACM.

Gupta, D., Jalote, P., and Barua, G. (1996). A Formal Framework For Online Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131.

Gurtner, D. (2011). *Safe Dynamic Software Updates in Multi-Threaded Systems with ActiveContext*. PhD thesis, University of Bern.

Hayden, C., Hardisty, E., Hicks, M., and Foster, J. (2009). Efficient Systematic Testing For Dynamically Updatable Software. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 9:1–9:5, New York, New York, USA. ACM.

Hayden, C., Magill, S., Hicks, M., Foster, N., and Foster, J. (2011a). Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the International Conference on Verified Software: Theories, Tools, and Experiments*, Philadelphia, Pennsylvania, USA. Springer-Verlag.

Hayden, C., Smith, E. K., Denchev, M., Hicks, M., and Foster, J. (2011b). Kitsune : Efficient , General-purpose Dynamic Software Updating for C.

Hayden, C. M., Smith, E. K., Hardisty, E. A., Hicks, M., and Foster, J. S. (2011c). Evaluating Dynamic Software Update Safety Using Systematic Testing. *IEEE Transactions on Software Engineering*, pages 1–41.

Hayden, C. M., Smith, E. K., Hicks, M., and Foster, J. S. (2011d). State Transfer for Clear and Efficient Runtime Upgrades. In *Proceedings of the 3rd International Workshop on Hot Topics in Software Upgrades*, Hannover, Germany. IEEE Press.

Hicks, M. (2005). *Dynamic Software Updating*. PhD thesis, University of Pennsylvania.

Hjálmtÿsson, G. and Gray, R. (1998). Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program. In *Proceedings of the 9th USENIX Annual Technical Conference*, pages 6–6, New Orleans, Louisiana, USA. USENIX Association.

Honda, K. (1993). Types for Dyadic Interaction. In *Proceedings of the 4th International Conference on Concurrency Theory*, number CONCUR'93, pages 509–523, Hildesheim, Germany. Springer-Verlag.

Honda, K., Kubo, M., and Vasconcelos, V. (1998). Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of the 7th European Symposium on Programming Languages and Systems*, volume 171, pages 122–138, Lisbon, Portugal. Springer-Verlag.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty Asynchronous Session Types.

Kaashoek, F. and Arnold, J. (2009). Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 187–198, Nuremberg, Germany. ACM.

Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 2nd edition.

Makris, K. and Bazzi, R. (2009). Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the 19th USENIX Annual technical conference*, pages 31–44, San Diego, California, United States. USENIX Association.

Marino, D. and Millstein, T. (2008). A Generic Type-and-effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 39–50, Savannah, Georgia, USA. ACM.

Mostrous, D., Yoshida, N., and Honda, K. (2009). Global Principal Typing in Partially Commutative Asynchronous Sessions. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 316–322, York, UK. Springer-Verlag.

Neamtiu, I., Foster, J., and Hicks, M. (2005). Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Software Engineering Notes*, 30(4):1–5.

Neamtiu, I. and Hicks, M. (2009). Safe and Timely Dynamic Updates to Multi-threaded Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Dublin, Ireland. ACM.

Neamtiu, I., Hicks, M., Foster, J., and Pratikakis, P. (2008). Contextual Effects For Version-Consistent Dynamic Software Updating And Safe Concurrent Programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–49, San Francisco, California, USA. ACM.

Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. (2006). Practical Dynamic Software Updating for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, Ottawa, Ontario, Canada. ACM.

Nicoara, A., Alonso, G., and Roscoe, T. (2008). Controlled, systematic, and efficient code replacement for running java programs. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 - Eurosys '08*, (5005):233.

Nielson, F. and Nielson, H. R. (1999). Type and Effect Systems. pages 114–136. Springer-Verlag.

Orso, A., Rao, A., and Harrold, M.-J. (2002). A Technique for Dynamic Updating of Java Software. In *Proceedings of the International Conference on Software Maintenance*, pages 649–658, Montréal, Canada. IEEE Computer Society.

Scott, D. (1998). Assessing the Costs of Application Downtime.

Sewell, P. (2001). Modules, Abstract Types, and Distributed Versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–247, London, UK. ACM.

Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., Silva, D. D., Auslander, M., Krieger, O., Stumm, M., Ganger, G. R., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003). System Support for Online Reconfiguration. *Online*.

Stewart, D. and Chakravarty, M. (2005). Dynamic Applications from the Ground Up. In *Proceedings of the 9th ACM SIGPLAN Workshop on Haskell*, pages 27–38, Tallinn, Estonia. ACM.

Stoyle, G. (2006). *A Theory of Dynamic Software Updates*. Phd, University of Cambridge.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. (2005). Mutatis Mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4):183–194.

Subramanian, S. (2010). *Dynamic Software Updates : A VM-Centric Approach*. PhD thesis, University of Texas at Austin.

Subramanian, S., Hicks, M., and Mckinley, K. (2009). Dynamic Software Updates : A VM-centric Approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Dublin, Ireland. ACM.

Sun-Microsystems (2006). Java Request for Enhancement: Hot Code swapping. http://bugs.sun.com/view_bug.do?bug_id=4910812.

Tempero, E., Bierman, G., Noble, J., and Parkinson, M. (2008). From Java to UpgradeJ: an empirical study. pages 1–5, Nashville, Tennessee. ACM.

Thomas, W., Wimmer, C., and Stadler, L. (2010). Dynamic Code Evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 10–19, Vienna, Austria. ACM.

Wernli, E., Gurtner, D., and Nierstrasz, O. (2011). Using First-class Contexts to realize Dynamic Software Updates. In *Proceedings of the 3rd International Workshop on Smalltalk Technologies*, pages 21–31, Edinburgh, Scotland. ACM.

Wikstrom, R. and Williams, M. (1993). *Concurrent Programming in Erlang.* Prentice Hall International (UK) Ltd., 2nd edition.

Yoshida, N. and Vasconcelos, V. (2006). Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited : Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93.

ZeroTurnaround (2011). JRebel. http://zeroturnaround.com/jrebel/.