UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

A progress report submitted for continuation towards a PhD

Supervisor: Dr Julian Rathke
Examiner: Professor Vladimiro Sassone

Behavioural Properties and Dynamic Software Update for Concurrent Programs

by Gabrielle Anderson

January 17, 2011

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

A progress report submitted for continuation towards a PhD

by Gabrielle Anderson

Correctly developing multi-threaded programs is notoriously difficult, and getting total coverage using traditional testing paradigms, to guarantee the program is correct, is often infeasible. We expand on previous work to provide various tools, namely a generalisation of session typing and an extension of policy automata to multi-threaded code, with which to verify multi-threaded code. Additionally, most programs are not written once and then left; maintaining and updating software is an essential part of the software development cycle. Dynamic software update (DSU) "is a technique by which a running program can be updated with new code and data without interrupting its execution" [45] and uses code analyses to ensure given safety properties are maintained across update boundaries. We present techniques for verifying if a modification can be applied to a running program whilst maintaining the desired behavioural properties, which may be those the program had before or some new properties.

Contents

List of Figures

Acknowledgements

Thanks to my supervisor Dr Julian Rathke for his ideas, suggestions for reading, and aid in making my writing intelligible, and to my adviser Dr Bernd Fischer. Thanks also to Ross Horne for all our discussions and debates, and to Dr Denis Nicole for his thoughts on concurrency and suggestions of concurrent application examples.

Chapter 1

Introduction

Correctly developing multi-threaded programs is notoriously difficult. On one hand there are the traditional problems of deadlock and livelock, which are general problems which most programmers wish to exclude. On the other hand are the actual properties which the program should have, such as communicating according to a protocol or maintaining separation of information which would lead to a conflict of interest. Additionally, whilst establishing behavioural properties for software is necessary, software also requires updates. It may be to fix bugs, add functionality, or simply make the system more efficient but software requirements are rarely static and systems are never perfect. In this thesis we present two approaches for defining behavioural properties for multi-threaded programs, and DSU techniques to permit modifying code at runtime whilst maintaining these properties.

In a realistic system, in order to safely use the values returned when accessing shared resources we must know something about their type. Traditionally values obtained from shared resources such as channels and shared memory have either been treated as top or are assumed to be of one specific type [32]. Work on session typing has pioneered an approach to allow values of different types to be safely sent and received on a channel [10,21,28,29,43,46,48]. We expand this formalism to generalised resources and resource accesses, where a resource access can return values of different types. We expose the essence of prior formalisms and proofs and discuss how we believe this approach can simplify the myriad different proofs for different session typing systems.

There are many behavioural properties which can be described using policy automata. These automata decide whether the side effects of a program conform to some programmer defined property. Examples of such properties include correct use of mutual exclusion, prevention of spoofing attacks by web servers, enforcing Chinese Wall properties, and prevention of denial of service attacks. We extend previous work on local access policies [3–7] to multi-threaded programs by representing multi-threaded effects using trace-equivalent single-threaded effects. We also exploit the insight that we need only model check for safety with respects to some policy up until the point an execution becomes blocked, which can significantly reduce the space which needs to be model checked.

The standard method used to update software is to stop running the code, change the code in an arbitrary manner, then restart the program. This methodology is at best intrusive and disruptive and at worst catastrophic. The restart causes an interruption to service which can simply be inconvenience, for example when using a PC, but in some systems is simply unacceptable. Additionally, even if it is possible to take down the service for updating, the changes are made in an arbitrary manner and there are no guarantees the system will come back online or will continue to have the desirable behaviour. Hence stronger guarantees about system behaviour, both in the process of application and the resulting system, are required for system safety.

An example of when it would have been desirable to update a system but it was impossible since it would have interrupted a non-redundant service is the ESA Cassini-Huygens probe to Titan. After launch a programmer error in software was discovered that prevented one of the two receivers on Cassini from being used. This prevented the orbital Cassini from receiving data from the Huygens probe, and a loss of half of the photos taken by the probe. It was not possible to take down the control software running the orbiter as it would have risked careening out of control. However if it had been possible to update the software whilst it was still running it might have been possible to return the orbiter to full functionality.

The above example demonstrates what problems can occur when one cannot modify running software or when one makes unconstrained changes. Whilst it is possible to write unconstrained updates which modify live code and don't cause errors, to do so consistently requires great skill on the part of the programmer and often is mere serendipity. Hence in order to free the programmer from such accidental complexity [15] we can develop static analyses (generally included in compilers) which indicate when such errors will probably happen. Dynamic software update (hereafter referred to as DSU) often makes use of formal approaches to modifying software whilst it is still running. The goal of DSU is to permit modification of software without shutting down the system, and guaranteeing that some safety property of the system which held before the update holds after the update. This removes for the programmer the complexity of having to worry about the safety of her patch, as such an analysis will guarantee it.

In several papers [13, 26, 37–39, 44] the authors present formal analyses with simple type safety guarantees for DSU for both single-threaded and multi-threaded execution. Designing multi-threaded systems is non-trivial as the standard problems for such systems include synchronisation issues, deadlock and race conditions. We hence extend DSU to behavioural properties which are crucial for multi-threaded program development.

This thesis is structured as follows. In Chapter 2 we present background material and related literature. In Chapter 3 we present resource access with variably typed return, which is a generalisation of session typing and message passing systems. This work reveals the essence of session typing analysis and we hope it will lead to simpler proofs for related systems. In Chapter 4 we expand work on local access policies for side effecting systems from single-threaded to multi-threaded programs. We show how to represent multi-threaded effects in trace equivalent single-threaded effects and how to reduce the search space by ignoring blocked traces. In Chapter 5 we present our work on dynamically updating multi-threaded programs whilst preserving their behavioural properties. We define a runtime safety analysis and outline how we intend to expand the work to a more efficient, static analysis. We conclude in Chapter 6.

Chapter 2

Background Material and Literature

2.1 Static Analyses and Type Systems

The concept of safety has already been introduced in passing. Pierce suggests that:

> Safety refers to the language's ability to guarantee the integrity of [machine service] abstractions and of higher-level abstractions introduced by the programmer using the definitional facilities of the language. [41]

There are many such abstractions ranging from synchronisation and mutual exclusion primitives to function application to array access depending on the language being discussed. It is desirable to maintain such abstractions, and there are several methods used to try to guarantee safety.

Some languages provide no guarantees that a particular program adheres to such abstractions. These are generally un-typed languages or languages where the type system can be 'switched off'. Writing a program in such a language and expecting it to be safe (using the standard meaning of type safety) corresponds to the hypothetical approach to DSU of hand crafting updates and expecting them to safely modify code without requiring restarting or causing errors. It is possible to create safe updates in this way but it requires the programmer to keep many low level details in her head and to design her program accordingly. This is not a methodology which will lead to effective or efficient software engineering practices.

Safe languages perform analyses to ensure that the abstractions are respected. These analyses can either be performed at compile time or at runtime, thence being known as static and dynamic analyses respectively. Dynamic analyses are less conservative than static ones (with regards to which safe programs they reject) as they analyse runtime data and values rather than abstract syntax trees. As dynamic analyses perform their checks at runtime they add runtime overhead. Static analyses perform their checks at compile time and hence will always be conservative, as determining control flow can in many languages be reducible to the Halting Problem. They do however permit errors to be detected and corrected before deployment of the software.

Type systems are generally static analyses and are described by Pierce as:

> A tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute. [41]

This methodology is generally reliant upon a program having structural operational semantics; this is the concept defined by Plotkin that the behaviour of an expression is defined completely by the composition of the behaviour of its sub-expressions [42]. Behaviour which is not defined in this way is labelled emergent, and is more difficult to reason about and often considered unsafe. An example of such behaviour is race conditions which, whilst they sometimes can be used in a reliable way, programmers often try to prevent. Such typing analyses generally will use this composability and structural inductive methods to guarantee safety properties.

It could be argued that dynamic analyses would be most appropriate in such a setting. These could provide a system which didn't have to restart to update and would never apply a corrupting update. They would also probably be less conservative with respects to which updates were rejected as unsafe. As a programming tool, however, they would be probably be less useful as the programmer would have to wait until she attempted to apply the update to the system before she discovered if the update was fundamentally incompatible with the system, and hence needed to be rewritten. Static analyses can examine the offline source code and the source of the update and determine whether the update would be always, occasionally or never compatible with the program. This information could then be brought to the attention of the programmer who could modify her update accordingly. Hence static analyses are an important tool in order to write updates for DSU.

2.2 Session Typing

Session typing is a family of static typing analyses which permit verification that a set of processes communicate according to a specified protocol. Traditionally session typing has focused on verifying dyadic interactions, which is of limited use in a multi-threaded system with an arbitrary number of threads. The formal underpinnings of session typing have been studied since the early 1990s [27, 28], though recently some errors in the foundational paper have been discovered and corrected [48].

Whilst the work in the majority of papers on session types uses π calculus style calculi [11, 27, 29, 35, 48], in some a λ calculus formulation is used [22, 23], which is closer to the conventional programming languages which would be employed in a DSU situation. In [22] the authors provide subtyping for session types, whilst in [23] the authors include a linear type system and permit aliasing of variables referring to channels. In neither paper do the authors satisfactorily handle functions with communicative side-effects as they only permit functions which make tail calls. This is in order to simplify the type and effect analysis as in such a case one does not have to account for nested behaviour. The work in [22, 23] provides a useful starting point for session types for more conventional languages.

In [11, 29] the authors generalise the session typing to sessions which can include more than two parties (multi-party session typing) in a π calculus paradigm. They also introduce the important concept of delegation, where an entity A can pass responsibility for performing certain actions within a session on to entity B. However the formulation of this in [29] was such that the delegation was not completely transparent. This could expose implementation details of the delegator to the delegate which could be either technically or commercially undesirable. This also would reduce generalisability and possible updates to systems, which would be especially problematic for DSU systems. In [11] however the authors deal with the above limitations and provide the first robust, multi-party, session type system. They also provide a new guarantee of global progress, that a well typed system will not experience deadlock part way through a session due to the communications.

The authors of [11, 29] introduce the concept of global session types. These specify in a high level manner the protocol according to which the participant (known as a role) must communicate. The global view can be projected down to the local view of each role and how it should communicate with the other roles. The local view is a session type, as defined in the majority of the session typing literature [11, 22, 23, 27, 29, 35, 48]. In this work the communications mechanisms are expanded to include multicast sending and channel transmission which is session delegation.

In a system where threads can interact we require additional safety properties, in order to prevent emergent behaviour. In a message passing system (with no shared memory) the only way which threads can interact with or affect each other is by their communications. Hence if we could prove safety properties with respects to the communications across update boundaries then we could update individual or groups of threads and prevent unsafe emergent behaviour.

2.3 Access Policies

In [32] the authors present a type system for ensuring that resources are accessed according to predefined patterns. This is a formulaically elegant and general analysis which could be used in many diverse situations. Hypothetically the analysis in [32], slightly modified to ensure complementary actions between roles, could be used as a session typing system. The analysis in [32] could also probably be used to control and validate shared memory access. This work, however, only considers single-threaded functional programs, and only permits specifying policies on single resources.

Extensive work has been done on validating correct use of resources with respect to local automata policies [3, 5–7]. This work permits policies to be specified with respects to multiple resources, which can be both static or dynamic (created at runtime). In [5] the authors present the concept of local policies, which only take account of specific actions on specific resources - all other actions are ignored. This approach allows a policy designer to focus on the property they are interested rather than the global state of available resources, as previous approaches have done [8]. In [3, 6, 7] the authors present a methodology for efficiently model checking the policies against the actions of a program. Of particular note is their approach of transforming resource usages into Basic Process Algebras and using a weakened form of model checking to make their approach complete as well as sound. The authors posit that their approach could be simply extended to multi-threaded programs by transforming into Basic Parallel Processes [18] rather than BPAs [7]. Recent work in equivalence decidability [30], however, shows that trace equivalence, on which their approach is based, is undecidable for BPPs. We instead look at representing the behaviour of multi-threaded programs as the behaviour of single-threaded programs, which can be transformed to BPAs as in [7].

A lot of other research has been done on usage policies and their enforcement mechanisms which is not relevant to this thesis. For a fuller review of the area see [3].

2.4 Dynamic Software Updating

In this section we present the relevant literature to DSU. Particularly we emphasise how the formal approaches in [37, 45] are thorough and suitably general for simple typing in single-threaded programs and multi-threaded programs respectively. We posit, however, that simple typing is insufficient to describe the safety properties often required for multi-threaded programs, and describe how no current work exists for dynamically modifying formal behavioural properties of multi-threaded code. We refer to multi-threaded DSU as MDSU.

An early example of DSU is Erlang [2]. Erlang is a dynamically typed language designed for distributed and concurrent programs in telephone systems which often need to be

Main Loop **Original Function Definitions: $f_1(), g_1()$** **Function Definitions After Update: $f_2(), g_2()$**

```
main
  do
    ...
    g()
    ...
    update
    ...
    f()
    ...
  loop
```

```
f() =
   ...
   update_log()
   ...

g() =
   ...
```

```
f() =
   ...

g() =
   ...
   update_log()
   ...
```

FIGURE 2.1: Update causing skipped log entry

modified. Its programs can have up to two versions of a module loaded, ‘new’ and ‘old’. Running code can refer to either the ‘new’ or the ‘old’ version, but by default refers to the ‘new’ version. If a newer version comes in any thread referring to the ‘old’ module is terminated (as the previous ‘new’ becomes the current ‘old’ and the previous ‘old’ is simply unloaded). Updates in Erlang are made safe by dynamic analyses which are not formally defined.

The work in [12] is the first attempt to apply static analysis techniques to DSU. The authors present a first-order, simply-typed, call-by-value lambda calculus, with the addition of a module system and an update primitive. In many ways the work is a generalisation of the dynamic module system used in Erlang in that the programmer can introduce new versions of modules and that by default code refers to the most recent version. There is however no limit on how many versions of a module the system can have loaded, and code can make explicit which module versions it is willing to use. The authors present an update system which guarantees type safety of code using different module versions.

This is a useful starting point to formally approaching DSU, particularly the infrastructure to permit the programmer to use multiple versions of a module. The granularity of an update is the module, which means that by an update one swaps in a new module (but a single update can modify multiple modules). A module can encapsulate many functions, and it seems slightly spurious to have to provide code for all the functions in a new module when only actually modifying one or two functions (and having to supply duplicate code for those functions which weren't modified). However with suitable tool support this could be automated.

In [13] the authors build on [12] focusing particularly on implementation and compiler tools but, whilst they present interesting motivational concepts behind [13], they provide no new formal technical approaches and rely on dynamic type checking in dynamic linking of verifiable native code. The work in [20, 36] is similar in providing case studies and compiler tools.

The next development to the static analysis approach to DSU is [45]. The authors present a C like language, including state, pointers and records (which can be use to implement structures) and facilities to dynamically update code. The granularity of update in this paper has changed from the module in [12] to the function - hence the programmer can change the function body (and type signature) of any function and the system will ensure that the updated code is still type safe. A single update can still modify more than one function. This system also permits updates to modify an abstract data type and hence the data items of the type being modified must also be modified so that they conform to the new type. Hence in this system when modifying an abstract data type the programmer must include a function to transform values from the old type to the new type.

The analysis in [45] still includes a dynamic aspect as the system, upon reaching an update command, it performs a dynamic check (using compile time type information) to verify that the update is valid at that specific update point. It would be possible to, at compile time, ensure that the update would be valid at every update point (a fully static analysis) using the type system of [45]. Instead the very last check at runtime so that if the update is valid at some update points but not at all the the update is not rejected as unsafe. Hence, whilst being in part a dynamic analysis, it could equally validly (and more conservatively) be constructed as a static analysis.

Implementation and compiler tool details for the work in [45] are provided in [39]. Together [39, 45] provide a comprehensive technique for dynamically updating single-threaded code with arbitrary changes to code, function definitions, type definitions and state. Since the work in [45] permits arbitrary updates with few restrictions, subsequent work on single-threaded DSU does not focus the generality of updates (increasing what can be updated). In [38] the authors instead focus upon making updates easier for the programmer to reason about.

In [13, 14, 45] updates occur when a specific language primitive, the *update* command, is encountered. These are written in the code by the programmer and, incorrectly placed, could cause updates to occur in a manner which whilst type safe would violate the constraints of how the programmer desires the system to behave. An example of such a violation is as follows. Consider the program in Figure 2.1 and a type safe update update changing the bodies of functions `f(),g()` from $f_1(), g_1()$ to $f_2(), g_2()$. In an unrolling of the main loop, the update is applied at the *update* command. This changes the system so that the code executed when calling `f()` and `g()` is $f_2()$ and $g_2()$ respectively. One effect of this is that after the update the function call to `update_log()` is no longer made in `f()` but in `g()`. Hence, since in this loop unrolling we have called the old version of `g()`, $g_1()$, and we will call the new version of `f()`, $f_2()$, `update_log()` is not called at all in this loop unrolling.

In this example `update_log()` appends a new entry to the log, and we are assuming the programmer expects a log entry to be written for each loop unrolling. As we showed above, however, in the loop unrolling when we apply the update then there will be no call to `update_log()`. Whilst we have not violated any type safety properties, since the update is type safe, the programmer's conceptual constraint of `update_log()` being called on each loop unrolling has been violated.

In [38] the authors present a DSU system whose main goal is to "[let] programmers reason more easily about the safety of updates". To achieve this *update* commands are no longer used, and they address the issue of when updates should occur using transactional techniques from database research. In their language the programmer can delineate regions of code inside which she does not wish an update to the code to (visibly) occur. The authors include a safety property (transactional version consistency) so that such a region of code will run entirely with old version code or entirely with new version code. Consider again the above example, with the body of the loop delineated as such a region. In the loop unrolling when the *update* occurs the old version of `g()` and the new version of `f()` are called. Hence the above example does not have transactional version consistency.

This analysis can be used to prevent errors of the type demonstrated in Figure 2.1. It is also of use for detecting thread shared locations, and hence the words 'Safe Concurrent Programming' in the title. It is, however, only applicable to DSU for single-threaded programs. The contribution to DSU is mainly one of a different method for the programmer to specify where updates should occur. The contribution's main motivation, and the issue addressed, is usability and hence makes little advance to the theoretical contribution to DSU. The work, however, is more liberal with respects to when and where updates may occur. In DSU we often require synchronisation between threads to perform an update. Hence each thread being less prescriptive about when and where an update occurs may make it easier to find a synchronisation point.

In [37] the transactional approach to updating programs is used to extend simple typing to MDSU. The authors make us of the key insight that, as they can automatically infer regions where a modification will make the program run with either entirely with old version code or entirely with new version code, whenever a set of threads are all in such a region they can all apply an update without having to synchronise. We make use of this technique when attempting to remove synchronisation requirements.

In [24] the authors move towards considering behavioural properties for multi-threaded programs. Instead of proving formal properties or behaviours, however, the authors take the industry standard approach of performing a set of tests on the program. The authors develop an approach to automatically generate tests which will ensure that regression tests which held before the update hold after it. If this were done naively, the number of update tests generated would be infeasibly large. The authors present an approach to significantly reduce the number of tests required.

To date several practical, formal, static analyses have been developed for DSU [14, 37, 38, 45]. Using formal techniques they ensure several safety properties (primarily simple type safety) are preserved across type boundaries. These analyses, particularly the work in [37,45], provide a comprehensive and safe system for performing general updates which preserve simple type safety. There are many standard concurrent behavioural properties, however, which are not covered by simple typing. Hence we intend on addressing such behaviours with additional analyses.

Chapter 3

Resource Access with Variably Typed Return

3.1 Introduction

In real world systems we need to be able to obtain (and type-safely use) meaningful values from shared resources, such as a message sent from another thread or a piece of mobile code to execute. Most systems will require a runtime type check to guarantee an obtained value is of the expected type.

Some static analyses take account of this problem, so that runtime checks are unnecessary. Work on general resource usage [31] effectively ignores the problem by making impure accesses return a value with a constant type, such as unit or a boolean to indicate whether the access was successful. Session typing analyses for message passing systems statically guarantee that values received on channels are of the annotated, expected type. These analyses are similar in concept, but each time any change is made to the semantics of the code or the message passing the safety proofs must be redone, often in radically different ways.

We extract and generalise the principle features of systems where resource accesses can return different types, using message passing and session typing as a base (Section 3.2). We proceed to similarly generalise the concepts used in session typing to prove type safe accesses. We crucially separate the aspects of the safety proof which can be done for general systems from those which are dependent on the semantics of accessing the resources, referred to as the *invariability proof* (Section 3.3). Finally we give an example of an invariability proof. We also discuss our intention to explore invariability proofs for different semantics and show how it is easier to work with invariability proofs than redoing proofs for each new system (Section 3.4).

$$
\begin{array}{rcl}
e & ::= & v \\
 & | & \texttt{rec}\, f{=}\lambda x.e \\
 & | & e\; e \\
 & | & \mathsf{snd}(e,e) \\
 & | & \mathsf{rcv}(e)
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & n \\
 & | & b \\
 & | & () \\
 & | & c \\
 & | & x
\end{array}
\qquad
\begin{array}{rcl}
P & ::= & \langle e \rangle \\
 & | & P \| P \\
\sigma & ::= & c{\mapsto}q \\
 & | & \sigma,\sigma \\
q & ::= & v \\
 & | & q,q
\end{array}
$$

FIGURE 3.1: Message passing language

$$
\frac{[\sigma]\, e_1 \longrightarrow [\sigma']\, e_1'}{[\sigma]\, e_1\, e_2 \longrightarrow [\sigma']\, e_1'\, e_2}\ (\text{RAppOne})
\qquad
\frac{[\sigma]\, e_2 \longrightarrow [\sigma']\, e_2'}{[\sigma]\, v\, e_2 \longrightarrow [\sigma']\, v\, e_2'}\ (\text{RAppTwo})
$$

$$
\frac{}{[\sigma]\, \texttt{rec}\, f{=}\lambda x.e\; v \longrightarrow [\sigma]\, e[\texttt{rec}\, f{=}\lambda x.e/f][v/x]}\ (\text{RAppThree})
$$

$$
\frac{[\sigma]\, e_1 \longrightarrow [\sigma']\, e_1'}{[\sigma]\, \mathsf{snd}(e_1,e_2) \longrightarrow [\sigma']\, \mathsf{snd}(e_1',e_2)}\ (\text{RSnd1})
\qquad
\frac{[\sigma]\, e_2 \longrightarrow [\sigma']\, e_2'}{[\sigma]\, \mathsf{snd}(v,e_2) \longrightarrow [\sigma']\, \mathsf{snd}(v,e_2')}\ (\text{RSnd2})
$$

$$
\frac{\sigma' {=} \sigma[c{\mapsto}\sigma(c),v]}{[\sigma]\, \mathsf{snd}(c,v) \longrightarrow [\sigma']\, ()}\ (\text{RSnd3})
$$

$$
\frac{[\sigma]\, e \longrightarrow [\sigma']\, e'}{[\sigma]\, \mathsf{rcv}(e) \longrightarrow [\sigma']\, \mathsf{rcv}(e')}\ (\text{RRcv1})
\qquad
\frac{\sigma(c){=}v,q \qquad \sigma'{=}\sigma[c{\mapsto}q]}{[\sigma]\, \mathsf{rcv}(c) \longrightarrow [\sigma']\, v}\ (\text{RRcv2})
$$

$$
\frac{[\sigma]\, e \longrightarrow [\sigma']\, e'}{[\sigma]\, \langle e \rangle \longrightarrow [\sigma']\, \langle e' \rangle}\ (\text{RProc})
\qquad
\frac{[\sigma]\, P_1 \longrightarrow [\sigma']\, P_1'}{[\sigma]\, P_1 \| P_2 \longrightarrow [\sigma']\, P_1' \| P_2}\ (\text{RParOne})
\qquad
\frac{[\sigma]\, P_2 \longrightarrow [\sigma']\, P_2'}{[\sigma]\, P_1 \| P_2 \longrightarrow [\sigma']\, P_1 \| P_2'}\ (\text{RParTwo})
$$

FIGURE 3.2: Message passing language semantics

3.2 Semantics of Simplified Shared Resources Systems

We look first at simplified message passing systems consisting of sending and receiving values. It's easier to see the essence of the impure semantics in this system, and it's easier to see what you need to prove to prove safety. After we've done that we look at how more complex behaviours fit in.

3.2.1 Semantics of Message Passing Systems

Message passing systems consist of shared channels where accesses to shared channels can add and remove values from the channel queues. We define a simplified message passing language and its resources in Figure 3.1 and its semantics in Figure 3.2. The pure aspects of the system are standard. The (impure) shared resources σ are a set of channel names c with associated queues q. Sending a value adds it to the end of a queue. Receiving a value takes a value off the front of the queue, if it exists.

Conceptually the impure aspects of a system can be broken down into four constituent parts: the structure of the shared resources, the accesses permitted on the resources, the mechanism by which we obtain return values when accessing resources, and the semantics of how accesses modify the resources.

For message passing systems the structure of the shared resources is channels and their associate channel queues, as in Figure 3.1. The permitted accesses to the resources are sending a value v on a channel c, $\mathsf{snd}(c, v)$, and receiving a value from channel c, $\mathsf{rcv}(c)$. The mechanism for obtaining return values when accessing resources are incorporated in the RSend and RRecv rules. We can abstract the mechanism out of the individual rules by defining a projection function on the channels which, for a receive, returns the value at the front of the channel queue (and blocks if it doesn't exist), and, for a send, returns a constant unit value:

$$\sigma(l) \overset{\text{def}}{=} \begin{cases} v & \sigma(c)=v,q \wedge l=\mathsf{rcv}(c) \\ () & l=\mathsf{snd}(c,v) \end{cases}$$

We then simply redefine the rules so that all resource accesses use this projection function:

$$\frac{\sigma'=\sigma[c\mapsto\sigma(c),v]}{[\sigma]\ \mathsf{snd}(c,v)\longrightarrow[\sigma']\ \sigma(\mathsf{snd}(c,v))}\ (\text{RSend'}) \qquad \frac{\sigma(c)=v,q \quad \sigma'=\sigma[c\mapsto q]}{[\sigma]\ \mathsf{rcv}(c)\longrightarrow[\sigma']\ \sigma(\mathsf{rcv}(c))}\ (\text{RRecv'})$$

The semantics of how accesses modify the resources is also written into the rules for resource accesses. Again we can abstract this semantics away from the main semantics of the language by defining a reduction relation:

$$\frac{\sigma'=\sigma[c\mapsto\sigma(c),v]}{\sigma\xrightarrow{\mathsf{snd}(c,v)}\sigma'}\ (\text{RSnd}) \qquad \frac{\sigma(c)=v,q \quad \sigma'=\sigma[c\mapsto q]}{\sigma\xrightarrow{\mathsf{rcv}(c)}\sigma'}\ (\text{RRcv})$$

and redefining the RSend and RRecv rules:

$$\frac{\sigma\xrightarrow{\mathsf{snd}(c,v)}\sigma'}{[\sigma]\ \mathsf{snd}(c,v)\longrightarrow[\sigma']\ \sigma(\mathsf{snd}(c,v))}\ (\text{RSend''}) \qquad \frac{\sigma\xrightarrow{\mathsf{rcv}(c)}\sigma'}{[\sigma]\ \mathsf{rcv}(c)\longrightarrow[\sigma']\ \sigma(\mathsf{rcv}(c))}\ (\text{RRecv''})$$

3.2.2 Semantics of Our General System

In order to obtain a generalised system which can be instantiated with different shared resources and semantics for accessing those resources we want to separate these impure aspects from the core language and semantics. This means that we will not have to redefine the language each time we wish to, for example, add a new access primitive or modify the semantics of an existing primitive.

The impure aspects of the system are encapsulated by the structure of the shared resources, the accesses permitted on the resources, the mechanism by which we obtain return values when accessing resources, and the semantics of how accesses modify the resources. We have shown how we can abstract the mechanism for return values by defining a projection function on the shared resources using the access we are performing, and also how we can abstract the semantics of primitives using a reduction relation

$$\frac{[\sigma]\, e_1 \longrightarrow [\sigma']\, e_1'}{[\sigma]\, e_1\, e_2 \longrightarrow [\sigma']\, e_1'\, e_2}\ (\textsc{RAppOne}) \qquad \frac{[\sigma]\, e_2 \longrightarrow [\sigma']\, e_2'}{[\sigma]\, v\, e_2 \longrightarrow [\sigma']\, v\, e_2'}\ (\textsc{RAppTwo})$$

$$\frac{}{[\sigma]\, \texttt{rec} f{=}\lambda x.e\, v \longrightarrow [\sigma]\, e[\texttt{rec} f{=}\lambda x.e/f][v/x]}\ (\textsc{RAppThree}) \qquad \frac{\sigma \xrightarrow{l} \sigma'}{[\sigma]\, \texttt{acc}^l \longrightarrow [\sigma']\, \sigma(l)}\ (\textsc{RAcc})$$

$$\frac{[\sigma]\, e \longrightarrow [\sigma']\, e'}{[\sigma]\, \langle e \rangle \longrightarrow [\sigma']\, \langle e' \rangle}\ (\textsc{RProc}) \qquad \frac{[\sigma]\, P_1 \longrightarrow [\sigma']\, P_1'}{[\sigma]\, P_1 \| P_2 \longrightarrow [\sigma']\, P_1' \| P_2}\ (\textsc{RParOne}) \qquad \frac{[\sigma]\, P_2 \longrightarrow [\sigma']\, P_2'}{[\sigma]\, P_1 \| P_2 \longrightarrow [\sigma']\, P_1 \| P_2'}\ (\textsc{RParTwo})$$

FIGURE 3.3: General Language Semantics

over resources and the performed access. We combine these into a single resource access primitive $\texttt{acc}^l$ (as in [31]) which performs the access l on the shared resources. Then all other access commands are simply syntactic sugar:

$$\textsf{snd}(c,v) \overset{\text{def}}{=} \texttt{acc}^{send(c,v)} \qquad \textsf{rcv}(c) \overset{\text{def}}{=} \texttt{acc}^{recv(c)}$$

For the core system we use a multi-threaded higher order lambda-calculus with recursive functions. The language expressions and threads are defined in Figure 3.4. The operational semantics are as defined in Figure 3.3. The pure semantics of the operational semantics of this language are standard.

$$\begin{array}{rl} e ::= & v \\ | & \texttt{rec} f{=}\lambda x.e \\ | & e\, e \\ | & \texttt{acc}^l \end{array} \qquad \begin{array}{rl} v ::= & n \\ | & b \\ | & () \\ | & r \\ | & x \end{array} \qquad \begin{array}{rl} l ::= & \alpha(v_1,\ldots,v_n) \\ & \\ P ::= & \langle e \rangle \\ | & P \| P \end{array}$$

FIGURE 3.4: Expression, Label, and Program Thread Grammars

In our abstraction we permit arbitrarily defined shared resources σ. We permit arbitrary accesses l to the shared resources, and use a single accesses primitive $\texttt{acc}^l$. The RAcc reduction modifies the accompanying shared resources according to the resource modification relation $\sigma \xrightarrow{l} \sigma'$. We do not define this reduction relation as the definition will change depending on the system. Instead we place requirements on the resources projection function and the resources reduction relation:

If $\sigma \xrightarrow{l} \sigma'$ then $\sigma(l)$ is defined.
$\sigma(l)$ is always a value.

We refer to the projection $\sigma(l)$ as a *resource access*. The first property requires that if we can perform a reduction using an effect label a resource access using that label is defined. The second property requires that the expression returned by a resource access is a value. We will discuss the significance of this property in Section 3.3.2. The return value of the RAcc reduction is defined as the projection of the resources using the access label.

In order to verify that a value returned from a shared resource access is used in a type safe manner we need to know what the type of that value is. In some simple systems all values returned by performing the same type of access on the same resource, e.g. receive actions on the same channel, are of the same type. An example of such a system is one which uses simply-typed communications channels, such as integer channels on which only integers can be sent and received. In these systems we can, at a resource access and without any global information, determine the type of the value returned and hence can verify type safe usage using simple typing.

When values of different types can be returned by performing the same type of access on the same resource it is more difficult to locally determine the return type. This often occurs when the type of the value returned is dependent on the current state of the shared resources, for example in a message passing system where sending and receiving of values of any type on a channel is permissible.

In these systems type usage errors can occur when the value obtained from a resource access is not of the expected type. For example, using the message passing system defined in Section 3.2.1, the following code will cause an error:

$$\begin{aligned}
&[c \mapsto \emptyset]\,\mathsf{snd}(c, \mathtt{true}) \parallel (\mathsf{rcv}(c) + 1)\\
&\quad\to [c \mapsto \mathtt{true}]\,() \parallel (\mathsf{rcv}(c) + 1)\\
&\quad\to [c \mapsto \emptyset]\,() \parallel (\mathtt{true} + 1)\\
&\quad\not\to
\end{aligned} \tag{3.1}$$

Such systems are not, however, inherently flawed. The existence of a type error is dependent on the resource accesses, the impure semantics, and the starting state of the resources. For example, with the same code and different starting resources no error occurs:

$$\begin{aligned}
&[c \mapsto 2]\,\mathsf{snd}(c, \mathtt{true}) \parallel (\mathsf{rcv}(c) + 1)\\
&\quad\to [c \mapsto 2, \mathtt{true}]\,() \parallel (\mathsf{rcv}(c) + 1)\\
&\quad\to [c \mapsto \mathtt{true}]\,() \parallel (2 + 1)\\
&\quad\to [c \mapsto \mathtt{true}]\,() \parallel 3
\end{aligned} \tag{3.2}$$

In situations where we can determine the return type at the resource access without knowledge of other threads' behaviour and the state of the resources we say that we can determine the type *locally*. Otherwise we determine it *globally*.

$$
\begin{array}{rcl}
\varphi & ::= & c!\langle T\rangle \\
& | & c?(T) \\
& | & \varphi;\, \varphi \\
& | & \epsilon
\end{array}
\qquad
\begin{array}{rcl}
\Phi & ::= & \varphi \\
& | & \Phi \parallel \Phi
\end{array}
$$

FIGURE 3.5: Effects for a simple session typing system

3.3 Proving Safe Access of Shared Resources

3.3.1 Type Safe Resource Usage for Message Passing Systems

Session typing analyses are a body of static analyses which have been designed to determine whether a program in a message passing system with untyped channels will have any type errors due to the values sent and received on the channels. Conceptually they do this as follows. They perform a type and effect analysis on the code, where the effect is an abstraction of the communication behaviour. They assume some starting state of the resources, usually channels with empty queues, or an empty state without channels which are dynamically created. Then they define some predicate over the effects which, when true, in combination with their knowledge of the impure semantics of the system and the starting state, can be proved to imply that no type errors occur.

We can do a sketch of such a system and proof. We define the effects in Figure 3.5, where $c!\langle T\rangle$ denotes sending a value of type T on channel c, and $c?(T)$ denotes receiving a value of type T on channel c. Note that instead of the effect label annotating the specific value sent, as in previous sections, that the type of that value is annotated instead. We assume some type and effect system which determines the effect for code in the message passing language from Section 3.2.1, so that, for example, we can determine that the code:

$$\mathsf{snd}(c_1, \mathtt{true});\, 4+5;\, \mathsf{snd}(c_2, \mathtt{true}) \parallel (\mathsf{rcv}(c_2)+1) \tag{3.3}$$

has the effect:

$$c_1!\langle \textsc{Bool}\rangle;\, c_2!\langle \textsc{Bool}\rangle \parallel c_2?(\textsc{Int}) \tag{3.4}$$

where $e_1;\, e_2$ is syntactic sugar for $\mathtt{rec} f = \lambda x. e_2\, e_1$ when x and f are free in e_2. We assume the resources start as:

$$\sigma = c_1 \mapsto \emptyset, c_2 \mapsto \emptyset$$

We assume that at most two threads make use of a channel. Then we need to guarantee that the values sent are of the type which the receiver expects. To do this we compare the effects of the threads which use each channel, and require that each effect is 'complementary' to the other [28]. We define the complementary predicate as follows:

$$\mathsf{compl}(\varphi_1,\varphi_2) \overset{\text{def}}{=} \begin{cases} \texttt{true} & \varphi_1{=}c!\langle T\rangle \wedge \varphi_2{=}c?(T) \vee \varphi_1{=}c?(T) \wedge \varphi_2{=}c!\langle T\rangle \\ \mathsf{compl}(\varphi_1',\varphi_2') \wedge \mathsf{compl}(\varphi_1'',\varphi_2'') & \varphi_1{=}\varphi_1';\varphi_1'' \wedge \varphi_2{=}\varphi_2';\varphi_2'' \end{cases}$$

This will then catch the error in effect (3.4), as the value sent is not of the type expected by the receiver.

If we did not make the restriction that channels are only used by two threads then the effects being complementary would not be a sufficient condition to prevent errors. Consider the following code and one of its possible evaluations:

Example 3.1.

$$\begin{aligned}
&[c \mapsto \emptyset]\,\mathsf{snd}(c,1);\ \mathsf{snd}(c,\textit{true}) \parallel \mathsf{snd}(c,2);\ \mathsf{snd}(c,\textit{false}) \parallel \mathsf{rcv}(c)+1;\ \neg\mathsf{rcv}(c)\\
&\quad\to [c \mapsto 1]\,\mathsf{snd}(c,\textit{true}) \parallel \mathsf{snd}(c,2);\ \mathsf{snd}(c,\textit{false}) \parallel \mathsf{rcv}(c)+1;\ \neg\mathsf{rcv}(c)\\
&\quad\to [c \mapsto 1,2]\,\mathsf{snd}(c,\textit{true}) \parallel \mathsf{snd}(c,\textit{false}) \parallel \mathsf{rcv}(c)+1;\ \neg\mathsf{rcv}(c)\\
&\quad\ldots\\
&\quad\to [c \mapsto 2]\,\mathsf{snd}(c,\textit{true}) \parallel \mathsf{snd}(c,\textit{false}) \parallel \neg\mathsf{rcv}(c)\\
&\quad\to [c \mapsto \emptyset]\,\mathsf{snd}(c,\textit{true}) \parallel \mathsf{snd}(c,\textit{false}) \parallel \neg 2\\
&\quad\not\to
\end{aligned}$$

In this situation, each of the senders sends values of the type which the receiver is expecting, but as there are multiple senders the sends can interleave resulting in a value of an unexpected type being received.

If we made different assumptions about the state of the starting resources we would need to ensure that, after receiving the messages already in the channel's queue, the effect of the receiving thread, ignoring these receives, is complementary to the effect of the other thread:

Example 3.2.

$$\begin{aligned}
&[c \mapsto 2]\,\mathsf{snd}(c,\textit{true}) \parallel \mathsf{rcv}(c)+1;\ \neg\mathsf{rcv}(c)\\
&\quad\to [c \mapsto 2,\textit{true}]\,() \parallel \mathsf{rcv}(c)+1;\ \neg\mathsf{rcv}(c)\\
&\quad\ldots\\
&\quad\to [c \mapsto \emptyset]\,() \parallel \neg\textit{true}\\
&\quad\to [c \mapsto \emptyset]\,() \parallel \textit{false}
\end{aligned}$$

Here the effect of the code is:

$$c!\langle\textsc{Bool}\rangle \parallel c?(\textsc{Int});\ c?(\textsc{Bool}) \tag{3.5}$$

If we discard the prefix $c?(\textsc{Int})$, which corresponds to receiving the integer value already in the queue, then the remaining effects $c!\langle\textsc{Bool}\rangle$ and $c?(\textsc{Bool})$ are complementary, which reflects the safety of the code with the specified starting resources.

3.3.2 Type Safe Resource Usage in our General System

In Section 3.3.1 we demonstrate how, given assumptions about the semantics of message passing systems, we can determine whether some code would safely use the resources. We also discuss how the safety of code can depend on the starting state of the resources and on the interleaving of accesses to the resources. In this section we will reduce our assumptions, and show how to prove type safety in a way which takes note of the initial resources and possible interleaving of accesses.

As in Section 3.2.2 we want to permit the structure of shared resource and the resource reduction relation to be defined arbitrarily (whilst still retaining the reduction relation properties stated). We require that the expression returned by the resource access is actually a value. This is to guarantee that the thing returned cannot be evaluated with an unspecified effect. If the value is a function then its effect is annotated on the type. Hence when we guarantee that the type of the returned expression does not vary we also guarantee that the effect of the function does not vary.

In general we cannot locally determine the type returned by a resource access; recall the Code 3.1 where the type returned depends on the interleaving of other threads' actions. Our novel approach is instead to infer locally the *expected type* of the value returned using the context. We then delay ensuring that the value actually will be of that type until we are typing threads in parallel with each other and we can reason about the global, interleaving, behaviour. We annotate the expected type of a resource accesses on the access itself; the code $\mathtt{acc}^{l}$ will have a simple type T and an effect (L, T). We refer to the thread in which an access occurs as the *containing thread*.

We define the type, effect, and label grammars in Fig 3.6. The effects are standard, with the addition of annotating the expected type on the effect with the effect label. Recursive effects are equi-recursive, i.e. $\mu\mathbf{t}.\varphi \equiv \varphi[\mu\mathbf{t}.\varphi/\mathbf{t}]$. As the code is concurrent we also define concurrent effects as a small process algebra Φ. We define our type and effect system in Figure 3.7. The rules for expressions are standard [40]; the effects abstract the side effecting behaviour of the code. We use a standard type and effect judgement $\varphi \wr \Gamma \vdash e \colon T$ which denotes that, given the type information Γ, the expression e has simple type T, and the side effect of evaluating the expression is conservatively described by the effect φ.

The underlying assumption we made about the semantics in Section 3.3.1 is that the values returned by receive accesses are the same values which were sent on the channel, and that the first value which was sent is the first one received. This permits us to simply look at the types of the values sent, and the order that they are sent in, and guarantee

$$
\begin{array}{rcl}
\varphi & ::= & (L, T) \\
& | & \varphi ; \varphi \\
& | & \mu \mathbf{t}.\varphi \\
& | & \epsilon
\end{array}
\qquad
\begin{array}{rcl}
\Phi & ::= & \varphi \\
& | & \Phi \parallel \Phi \\
& & \\
L & ::= & \alpha(T_1, ..., T_n)
\end{array}
\qquad
\begin{array}{rcl}
T & ::= & \textsc{Bool} \\
& | & \textsc{Int} \\
& | & \textsc{Unit} \\
& | & T \xrightarrow{\varphi} T
\end{array}
$$

FIGURE 3.6: Type and Effect Grammars

$$
\frac{}{\emptyset \wr \Gamma \vdash n \colon \textsc{Int}}\,(:\textsc{TInt})
\qquad
\frac{}{\emptyset \wr \Gamma \vdash b \colon \textsc{Bool}}\,(:\textsc{TBool})
\qquad
\frac{}{\emptyset \wr \Gamma \vdash () \colon \textsc{Unit}}\,(:\textsc{TUnit})
$$

$$
\frac{}{\emptyset \wr \Gamma \vdash r \colon \mathsf{Res}\, r}\,(:\textsc{TRes})
\qquad
\frac{x \colon T \in \Gamma}{\emptyset \wr \Gamma \vdash x \colon T}\,(:\textsc{TVar})
\qquad
\frac{\Gamma \vdash l \colon L}{(L, T) \wr \Gamma \vdash \mathsf{acc}^{l} \colon T}\,(:\textsc{TAcc})
$$

$$
\frac{\varphi \wr \Gamma, x \colon T_1, f \colon T_1 \xrightarrow{\varphi} T_2 \vdash e \colon T_2}{\emptyset \wr \Gamma \vdash \mathtt{rec} f = \lambda x.e \colon T_1 \xrightarrow{\varphi} T_2}\,(:\textsc{TLam})
\qquad
\frac{\varphi_1 \wr \Gamma \vdash e_1 \colon T_1 \xrightarrow{\varphi_3} T_2 \quad \varphi_2 \wr \Gamma \vdash e_2 \colon T_1}{\varphi_1; \varphi_2; \varphi_3 \wr \Gamma \vdash e_1\, e_2 \colon T_2}\,(:\textsc{TApp})
$$

$$
\frac{l = \alpha(v_1, ..., v_n) \qquad L = \alpha(T_1, ..., T_n) \qquad \emptyset \wr \Gamma \vdash v_i \colon T_i}{\Gamma \vdash l \colon L}\,(:\textsc{TLab})
$$

$$
\frac{\varphi_i \wr \emptyset \vdash e_i \colon T_i}{\vdash \langle e_1 \rangle \parallel ... \parallel \langle e_n \rangle \colon \varphi_1 \parallel ... \parallel \varphi_n}\,(:\textsc{TPar})
$$

$$
\frac{\vdash P \colon \Phi \quad \sigma \colon \Sigma \quad \mathsf{compatible}(\Phi, \Sigma)}{\sigma \vdash P \colon \Phi}\,(:\textsc{ValidThreads}) \quad \text{where} \quad \begin{array}{l} \mathsf{compatible}(\Phi,\Sigma) \\ \Rightarrow \mathsf{compGen}(\Phi,\Sigma) \end{array}
$$

FIGURE 3.7: Type Rules for Expressions, Labels, and Threads

that the receives are all of the same type in the same order. Such an assumption makes for reasonably easy proofs, but will obviously not always be true; we cannot necessarily rely a simple FIFO semantics of values added to the shared resource when using, for example, a distributed database.

3.3.2.1 Using types instead of values in effects

In the same vein as our approach to typing message passing we annotate the type of the argument value on the effect label, rather than the value itself. This permits us to perform arbitrary pure computation in the argument positions of access labels. If we did not permit this then we could only permit non-variable values in argument positions, which would be a sever restriction. As it is, we must introduce some minor additional restrictions on the impure semantics, which conceptually will permit us to perform our static analysis on an abstraction of the resources and their semantics rather than their concrete counterparts.

In addition to the two requirements stated in Section 3.2.2 we also require a resources abstraction Σ, its reduction relation $\Sigma \to \Sigma'$, its projection function $\Sigma(L)$, and a way to relate the two $\sigma\colon\Sigma$, which satisfy the following properties:

$$\begin{array}{l}\text{If } \sigma \xrightarrow{l} \sigma' \text{ and } \sigma\colon\Sigma \text{ then } \emptyset \vdash l\colon L \text{ and } \Sigma \xrightarrow{L} \Sigma'.\\ \text{If } \sigma\colon\Sigma \text{ and } \emptyset \vdash l\colon L \text{ then } \sigma(l)\colon\Sigma(L).\end{array}$$

3.3.2.2 Globally checking type safety

In order to guarantee type safety we need to verify that each resource access returns a value of the expected type, which is annotated on the effect of that resource access, irrespective of the previous accesses that have occurred. As the return type of a resource access will depend on the state of the shared resources we must, in the general case, explore all possible states that the shared resources can be in when a specific access is reached, and guarantee that the value returned for each state is of the expected type. Since we need to explore the state space of the resources, under the effect of the code, the initial state of the resources is important as it is the state which we will reduce.

Informally, for each access (L, T) we require that, irrespective of the occurrence or interleaving of other resource accesses which can occur before (L, T), the value returned by access an access of type L is of type T. When typing a lone thread this simply consists of considering how the resources are reduced using the effect of the thread before a given access. When typing multiple threads in parallel we must consider all interleavings of the effects on the state of the shared resources from parallel threads. If the type of a given access is equal to the expected type for that access irrespective of the scheduling of the resource accesses by threads in parallel or by the containing thread, up to the point of the given access, we refer to the set of threads as compatible.

To state this property formally we define a predicate compatibleGeneral (shortened to compGen) and the auxiliary function interleavings (shortened to int). The interleavings function provides all possible interleavings of a set of parallel effects. We also define a 'sub-run' relation $\varphi_2 \preceq \varphi_1$ which denotes that φ_2 is a prefix of one possible run of φ_1; at this point it is simply a prefix as an effect can only have one possible run, but when we introduce choice the relation will become more complex due to different possible runs for the same effect. The sub-run relation is the smallest relation defined over:

$$\frac{\varphi' \preceq \varphi}{\varphi' \preceq \varphi;\, \_} \qquad \frac{}{\varphi \preceq \varphi}$$

We define $\sigma \xrightarrow{\varphi}{}^{*} \sigma'$ as the reflexive and transitive closure of $\sigma \xrightarrow{(L,T)} \sigma'$.

$$\begin{array}{lcl}\mathsf{compGen}(\varphi_1,\ldots,\varphi_n,\Sigma) & \stackrel{\text{def}}{=} & \forall i{\in}1\ldots n, \varphi'_1{\preceq}\varphi_1,\ldots,(\varphi'_i;(L,T)){\preceq}\varphi_i,\ldots,\varphi'_n{\preceq}\varphi_n,\\ & & \forall\varphi{\in}\mathsf{int}(\varphi'_1,\ldots,\varphi'_n).\text{if there is a reduction}\\ & & \text{path } \Sigma\xrightarrow{\varphi}{}^{*}\Sigma'\xrightarrow{(L,T)}\Sigma'' \text{ then } \Sigma'(L){=}T\end{array}$$

$$\mathsf{int}(\varphi_1,...,\varphi_n) \quad \overset{\text{def}}{=} \quad \bigcup_{i=1}^{n}\{(L,T);\varphi'|\varphi_i{=}(L,T);\varphi'_i\wedge\varphi'{\in}\mathsf{int}(\varphi_1,...,\varphi'_i,...,\varphi_n,\})$$

The predicate compGen is a formalisation of model checking the possible interleavings of the effects on the resources σ. It includes a check for the existence of a reduction path as, depending on the resources reduction relation, we may not be able to perform all effects in any order. For example in a blocking asynchronous message passing system we cannot receive a message before it is sent. This approach has exponential complexity. This is, however, the general solution where we know nothing about the structure of the resources, the permissible accesses, and all we know about the resource reduction relation are the two properties from Section 3.2.2. Given more information about the resources reduction relation then we may be able to define a predicate compatible (otherwise known as an *invariability predicate*) such that:

$$\mathsf{compatible}(\Phi,\sigma) \quad \Rightarrow \quad \mathsf{compGen}(\Phi,\sigma)$$

As this strengthened predicate implies general compatibility we can employ instead of the general predicate, in situations where we can provide a proof of this implication. In the ValidThreads rule we use this predicate instead of compatibleGeneral. The complexity of this predicate will vary depending on the situation, but can be as low as linear complexity. We refer to proving such an implication as an *invariability proof*. We give examples of such proofs, and explore how they simplify the proof process, in Section 3.4.

3.3.3 More complex behaviours

Some of the strengths of message passing systems with session typing include permitting variable sessions to occur, where one thread can determine which paths the other threads take, and delegation, where one thread sends another thread a channel for it to transmit on. In this section we explore these properties and how they interact with proving type safe access.

3.3.3.1 Internal and external choice

In order to permit variable sessions message passing systems can include a pair of complementary constructs which correspond to internal and external choice (often referred to as selection and branching). Internal choice $c{\oplus}\langle e\rangle.e'$ consists of sending the value that e reduces to the dependent thread, via c, and then evaluating e' which contains the code for the chosen path. External choice $c\&(T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n)$ consists on performing a case split on the type of the value received by the dependent thread, via c, where e_1 contains the code for each possible path i. We extend the effects with $c \oplus \langle T\rangle.\varphi'$ and $c\&(T_1 \Rightarrow \varphi_1, ..., T_n \Rightarrow \varphi_n)$, which have a similar meaning.

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & | & \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \\
 & | & \texttt{acc}^l \& (T_1 \Rightarrow e_1, \ldots, T_n \Rightarrow e_n)
\end{array}
\qquad
\begin{array}{rcl}
T & ::= & \ldots \\
 & | & T \& T
\end{array}
\qquad
\begin{array}{rcl}
\varphi & ::= & \ldots \\
 & | & \varphi \oplus \varphi \\
 & | & \varphi \& \varphi
\end{array}
$$

FIGURE 3.8: Extended expressions, types and effects

In message passing systems there is an obvious dependency between the values sent on a channel and the values received from it, and hence between the access which is chooses the path and the access which receives that choice. We can express this by extending the 'complementary' relation:

$$
\mathsf{compl}(\varphi_1,\varphi_2) \overset{\text{def}}{=} \begin{cases} \ldots & \\ \exists i. T = T_i \wedge \mathsf{compl}(\varphi', \varphi'_i) & \varphi_1 = c \oplus \langle T \rangle . \varphi' \wedge \varphi_2 = c \& (T_1 \Rightarrow \varphi'_1, \ldots, T_n \Rightarrow \varphi'_n) \vee \\ & \varphi_1 = c \& (T_1 \Rightarrow \varphi'_1, \ldots, T_n \Rightarrow \varphi'_n) \wedge \varphi_2 = c \oplus \langle T \rangle . \varphi' \end{cases}
$$

This denotes that two effects are complementary if one of them presents a series of choices of paths to follow, the other chooses which path to follow, and the effect of the option and the choice are complementary.

In general there is not necessarily a connection between one thread choosing a path and other threads accepting that choice. We can, however, generalise the concept. The generalisation of internal choice is simply the $\texttt{if}\, e\, \texttt{then}\, e'\, \texttt{else}\, e''$ construct, where we permit e' and e'' to have different effects. The generalisation of external choice is similar to the construct in message passing systems. We define a construct $\texttt{acc}^l \& (T_1 \Rightarrow e_1, \ldots, T_n \Rightarrow e_n)$ which denotes that the containing thread will perform an access l and will perform a case split on the type of the value returned.

As we now permit a specific resource access to return different types to be valid, we need to extend our definitions of types and our definition of compatibility. We extend the types with external choice which denotes a union type. Our extensions are defined in Figure 3.8. We then provide additional reduction rules and type rules for the new constructs in Figure 3.9.

We define a type inclusion relation as the smallest reflexive, transitive relation defined over the rules:

$$
\frac{}{T \leq T} \qquad \frac{T \leq T'}{T \leq T' \& T''} \qquad \frac{T \leq T''}{T \leq T' \& T''}
$$

This relation denotes that a type is included in a union type constructed using that type.

When there is a choice, either internal or external, about the possible effect of some code then we can obtain different runs for the same single-threaded effect. We hence

$$\frac{[\sigma]\, e_1 \longrightarrow [\sigma']\, e_1'}{[\sigma]\,\texttt{if}\, e_1\,\texttt{then}\, e_2\,\texttt{else}\, e_3 \longrightarrow [\sigma']\,\texttt{if}\, e_1'\,\texttt{then}\, e_2\,\texttt{else}\, e_3}\ (\textsc{RIfOne}) \qquad \frac{}{[\sigma]\,\texttt{if}\,\texttt{true}\,\texttt{then}\, e_2\,\texttt{else}\, e_3 \longrightarrow [\sigma]\, e_2}\ (\textsc{RIfTwo})$$

$$\frac{}{[\sigma]\,\texttt{if}\,\texttt{false}\,\texttt{then}\, e_2\,\texttt{else}\, e_3 \longrightarrow [\sigma]\, e_3}\ (\textsc{RIfThree}) \qquad \frac{\sigma \xrightarrow{l} \sigma' \qquad \sigma(l) \colon T_i}{[\sigma]\,\mathsf{acc}^l\&(T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n) \longrightarrow [\sigma']\, e_i}\ (\textsc{RExtChoice})$$

$$\frac{\varphi_1 \wr \Gamma \vdash e_1 \colon \textsc{Bool} \quad \varphi_2 \wr \Gamma \vdash e_2 \colon T \quad \varphi_3 \wr \Gamma \vdash e_3 \colon T}{\varphi_1;\ (\varphi_2 \oplus \varphi_3) \wr \Gamma \vdash \texttt{if}\, e_1\,\texttt{then}\, e_2\,\texttt{else}\, e_3 \colon T}\ (\textsc{:TIfExpr})$$

$$\frac{\varphi_i \wr \Gamma \vdash e_i \colon T \quad \Gamma \vdash l \colon L}{(L, T_1);\ \varphi_1 \& ... \& (L, T_n);\ \varphi_n \wr \Gamma \vdash \mathsf{acc}^l\&(T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n) \colon T}\ (\textsc{:tExtChoice})$$

Figure 3.9: Extended operational semantics and type rules

also extend the sub-run relation with the following:

$$\frac{\varphi_1' \preceq \varphi_1}{\varphi_1' \preceq \varphi_1 \& \varphi_2} \qquad \frac{\varphi_2' \preceq \varphi_2}{\varphi_2' \preceq \varphi_1 \& \varphi_2} \qquad \frac{\varphi_1' \preceq \varphi_1}{\varphi_1' \preceq \varphi_1 \oplus \varphi_2} \qquad \frac{\varphi_2' \preceq \varphi_2}{\varphi_2' \preceq \varphi_1 \oplus \varphi_2}$$

Finally we make a small modification to the general compatibility relation, to define an acceptable return type as one which is included in the expected type:

$$\begin{aligned} \mathsf{compGen}(\varphi_1, ..., \varphi_n, \Sigma) \overset{\text{def}}{=} & \ \forall i \in 1...n, \varphi_1' \preceq \varphi_1, ..., (\varphi_i'; (L, T)) \preceq \varphi_i, ..., \varphi_n' \preceq \varphi_n, \\ & \forall \varphi \in \mathsf{int}(\varphi_1', ..., \varphi_n').\text{if there is a reduction} \\ & \text{path } \Sigma \xrightarrow{\varphi}{}^{*} \Sigma' \xrightarrow{(L,T)} \Sigma'' \text{ then } \Sigma'(L) \leq T \end{aligned}$$

With these extensions we can represent effects which are dependent on the values returned from resource accesses, and still guarantee type safe usage of values obtained from resource accesses.

3.3.3.2 Delegation

In order to permit one thread to control the impure behaviour of another some message passing systems include functionality called delegation, where one thread sends a channel to another, for the latter to use normally. In future work we intend to include dynamic resources (which are used in Section 4.3, based on [7]). Once we permit dynamic resource values to be sent over the shared resources, and dynamic resources are nominally typed, and as we require for the values received from a specific resource access to be of the same type, we effectively get delegation for free, as we statically determine the types received by resource accesses.

$$\frac{}{\varphi \xrightarrow{\epsilon} \varphi}\ (\text{:ESkip}) \qquad \frac{}{(L,T);\ \varphi \xrightarrow{(L,T)} \varphi}\ (\text{:EAcc}) \qquad \frac{i \in 1,2}{\varphi_1 \oplus \varphi_2 \xrightarrow{\epsilon} \varphi_i}\ (\text{:EIntChoice})$$

$$\frac{\varphi_i \longrightarrow \varphi_i' i \in 1,2}{\varphi_1 \& \varphi_2 \xrightarrow{(L,T)} \varphi_i'}\ (\text{:EExtChoice}) \qquad \frac{\Phi_1 \xrightarrow{(L,T)} \Phi_1'}{\Phi_1 \parallel \Phi_2 \xrightarrow{(L,T)} \Phi_1' \parallel \Phi_2}\ (\text{:EParOne}) \qquad \frac{\Phi_2 \xrightarrow{(L,T)} \Phi_2'}{\Phi_1 \parallel \Phi_2 \xrightarrow{(L,T)} \Phi_1 \parallel \Phi_2'}\ (\text{:EParTwo})$$

Figure 3.10: Effect Reduction Rules

3.3.4 Properties

We present the main properties of the general system, which are subject reduction and fidelity.

3.3.4.1 Subject Reduction

Informally, subject reduction states that any reduction preserves or decreases the type and effect of an expression [33]. We formalise reduction validity for effects as the smallest transitive relation defined over the rules in Figure 3.10.

This relation denotes which reductions on effects are valid. Most of the definitions are straightforward. ESkip describes reductions which don't perform resource accesses and don't perform dynamic modifications. EAcc describes reductions which perform resource accesses. EIntOne, EIntTwo, EextOne, and EExtTwo describe choosing one possible path and following it. EParOne and EParOne describe how the effects of parallel threads may change.

Theorem 3.1. ***Subject Reduction***

If $\vdash P : \Phi$ *and* $[\sigma]\, P \rightarrow [\sigma']\, P'$ *then* $\vdash P' : \Phi'$ *and* $\Phi \dashrightarrow \Phi'$

We provide a sketch proof of subject reduction as follows. We prove subject reduction by induction over the operational semantics. The most important case for guaranteeing type safety, specifically that the types of resource accesses are equal to the expected types irrespective of how the resources may be dynamically modified, is the parallel reduction case. Given a well typed set of parallel threads, by the compatibility property we know that the expected types are invariant irrespective of the scheduling of their effects. This is verified by performing model checking, or some predicate which, given additional information about the semantics of the resources, implies compatibility. Hence any reductions will not invalidate this property.

3.3.4.2 Fidelity

We also prove the soundness of the effect analysis. In the fidelity theorem we guarantee that the actual reductions on the resources are matched by reductions on the abstraction, that such reductions can only occur if they are expected in the effect, and that the modified resources stay consistent with the abstraction. This is a generalisation of the fidelity property for session typing systems [29]. To prove this property we define a reduction relation using the rules in Figure 3.10 on resources and parallel effects which denotes how an effect modifies the resources:

$$\frac{\sigma \xrightarrow{\varphi} \sigma' \qquad \Phi \xrightarrow{\varphi} \Phi'}{[\sigma]\,\Phi \xrightarrow{\varphi} [\sigma']\,\Phi'}\ (\textsc{EffectRed})$$

Theorem 3.2. ***Fidelity***

If $\vdash P \colon \Phi$ *and* $[\sigma]\,P \xrightarrow{\varphi} [\sigma']\,P'$ *then* $[\sigma]\,\Phi \xrightarrow{\varphi} [\sigma']\,\Phi'$

3.4 Exploring Invariability Proofs

In future work we intend to explore invariability proofs for different semantics and show how it is easier to work with them as opposed to redoing proofs for each new system. We intend on contrasting the semantics for blocking message passing systems, non-blocking message passing systems, and 'dynamically typed' stores. In this report we simply give the example of blocking message passing systems.

3.4.1 Examples

3.4.1.1 Blocking Message Passing Systems

We make use of the effect grammar of Figure 3.5:

$$\begin{array}{rcl} \varphi & ::= & c!\langle T\rangle \\ & \mid & c?(T) \\ & \mid & \varphi;\,\varphi \\ & \mid & \epsilon \end{array} \qquad \begin{array}{rcl} \Phi & ::= & \varphi \\ & \mid & \Phi \parallel \Phi \end{array}$$

and the resource definition and semantics from Section 3.2.1, modified to work on types:

$$\Sigma(L) \overset{\text{def}}{=} \begin{cases} T & \Sigma(c){=}T,q \wedge L{=}c?(\_) \\ () & L{=}c!\langle T\rangle \end{cases}$$

$$\frac{\Sigma'{=}\Sigma[c{\mapsto}\Sigma(c),T]}{\Sigma \xrightarrow{c!\langle T\rangle} \Sigma'}\ (\text{TRS}\textsc{nd}) \qquad \frac{\Sigma(c){=}T,q \qquad \Sigma'{=}\Sigma[c{\mapsto}q]}{\Sigma \xrightarrow{c?(\_)} \Sigma'}\ (\text{TRR}\textsc{cv})$$

$$\frac{\sigma \xrightarrow{c!\langle T\rangle} \sigma'}{[\sigma]\, c!\langle T\rangle \rightarrow [\sigma']\, \sigma(c!\langle T\rangle)}\ (\text{RSEND''}) \qquad \frac{\sigma \xrightarrow{c?(T)} \sigma'}{[\sigma]\, c?(T) \rightarrow [\sigma']\, \sigma(c?(T))}\ (\text{RRECV''})$$

In order to make an invariability proof, we make use of the essence of session typing work: complementary effects [27]. As discussed in Section 3.3.1, this assumes that only two threads make use of a channel, and the effect of each on the channel is complementary to the effect of the other on the channel. We make use of the following formalisations: threadsUsing singles out the threads which use a particular channel, effectOn extracts the behaviour on one specific channel, and compl defines if two effects are complementary.

$$\mathsf{threadsUsing}(\Phi,c) \overset{\text{def}}{=} \begin{cases} \emptyset & c \notin \varphi \wedge \Phi = \varphi \\ \varphi & c \in \varphi \wedge \Phi = \varphi \\ \mathsf{threadsUsing}(\Phi_1,c) \cup \mathsf{threadsUsing}(\Phi_2,c) & \Phi = \Phi_1 \| \Phi_2 \end{cases}$$

$$\mathsf{effectOn}(\varphi,c) \overset{\text{def}}{=} \begin{cases} \varphi & \varphi = c!\langle\_\rangle \vee \varphi = c?(\_) \\ \epsilon & \varphi = c'!\langle\_\rangle \vee \varphi = c'?(\_) \vee \varphi = \epsilon \\ \mathsf{effectOn}(\varphi_1,c); \mathsf{effectOn}(\varphi_2,c) & \varphi = \varphi_1 ; \varphi_2 \end{cases}$$

$$\mathsf{compl}(\varphi_1,\varphi_2) \overset{\text{def}}{=} \begin{cases} \mathtt{true} & \varphi_1 = c!\langle T\rangle \wedge \varphi_2 = c?(T) \vee \varphi_1 = c?(T) \wedge \varphi_2 = c!\langle T\rangle \\ \mathsf{compl}(\varphi_1',\varphi_2') \wedge \mathsf{compl}(\varphi_1'',\varphi_2'') & \varphi_1 = \varphi_1' ; \varphi_1'' \wedge \varphi_2 = \varphi_2' ; \varphi_2'' \end{cases}$$

We can then define an invariability predicate for the blocking message passing system:

$$\mathsf{compatible}(\Phi,\sigma) \overset{\text{def}}{=} \forall c \in \Phi.\mathsf{threadsUsing}(\Phi,c) = \{\varphi_1,\varphi_2\} \wedge \mathsf{compl}(\mathsf{effectOn}(\varphi_1,c),\mathsf{effectOn}(\varphi_2,c))$$

Proving that this implies general compatibility is then simple. If the predicate is true, only two threads are using a shared resource. As their effects are complementary each send appends to the end of the message queue the type expected by the receiver. As the receiver blocks on a receive action when the queue is empty, the value received will always be of the correct type.

3.5 Conclusions

We show how obtaining values of the correct type from shared resource is a significant behavioural property for multi-threaded programs. We generalise the approach taken in session typing and discuss how in general, to prove safety, we need to perform some form of model checking. We discuss how, given more knowledge about a system, we can use predicates which are linear in effect size rather than model checking which is exponential. We present one such predicate and suggest future work comparing it with predicates for other systems, positing that defining such predicates and proving that they imply the general property is simpler than repeatedly doing session typing style proofs for each different system.

Chapter 4

Access Policies

4.1 Introduction

Both for program correctness and for security, it is essential that a program access resources in a valid manner. Validity of access may comprise guaranteeing that threads communicate according to a certain protocol, that code cannot access data sets of companies between which there is a conflict of interest, or that denial of service attacks cannot occur. Any time that we are interested in how data is used, moved, or viewed, we can consider it a resource, alongside traditional resources such as mutexes and network ports.

Many techniques have been developed in order to define and control how resources are used. We employ local usage automata to specify what sequences of resource accesses are invalid. Local usage automata formalise and enhance the concept of a sandbox. These automata are designed so that policies only specify the behaviour which we are interested in; other resource accesses are assumed to be on self loops within the automata. This approach is more flexible than global policies or explicit local checks in program code. Due to the local nature of the policies different pieces of code with different usage requirements can easily be composed. They also are designed so that policies can be applied to code which has been obtained from remote or untrusted sources. Finally these policies are parametric and do not have to be defined solely on statically known resources.

Whilst proving security and behavioural properties is important for single-threaded programs, it is arguably even more important for concurrent programs, where interleaving of accesses can lead to unforseen interactions. Previous work on policy automata has been limited to single-threaded programs. Whilst the authors of prior work on policy automata suggest that extending policy automata to concurrent programs would be simple, using BPPs [7], recent work shows this to be impossible [30].

Instead, we provide an intuitive extension of the calculus of usages presented in [7] (hereafter referred to as *single-threaded usages*) to include concurrency (hereafter referred to as *concurrent usages*). When model checking policy automata we check that the possible traces which a concurrent usage can generate respect the usage policies. In order to generate the approach for concurrent usages we make use of the insight that if some single-threaded usage is trace equivalent to some concurrent usage then the concurrent usage is safe if and only if the single-threaded usage is safe with respects to some automata. We show how extract a trace equivalent single-threaded usage for any concurrent usage. We prove this extraction process to be well defined and finite, for the finite systems that we are considering. We can then use existing model checking infrastructure [7] to ensure that the concurrent usage adheres to the desired access policies.

One of the issues when model checking concurrent programs is the that the state-space is exponential size with respects to the single-threaded usages which are in parallel. In order to reduce the space we make use of the fact that many concurrent systems include actions which can be blocked, depending on the state of shared resources. If an action is blocked then we can ignore an interleaving trace that we are model checking from that point onwards, as it will not occur at runtime. We further reduce the cost of model checking by making use of policy automata which simulate the semantics of the blockable resources, which means that we needn't run a model of the resources alongside the model checker in order to discover when a trace becomes blocked.

Our contribution is the following. We define a projection function which, for any concurrent usage, gives a trace-equivalent single-threaded usage Crucially, we prove that this projection is both well defined and finite. We use this projection function to extend model checking of local policy automata to concurrent systems. We show how to improve the efficiency and accuracy of this model checking process by eliminating traces which will not occur at run-time, using information about the semantics of the impure shared resources. This approach, naively, requires us to run a model of the semantics alongside the model checker. We show how to reduce this cost by simulating the impure semantics in the policy automata.

This paper is structured as follows. We summarise related work in Section 4.2. We describe the local usage policy approach for single-threaded programs [3,6,7] in Section 4.3. We extend access policies to multi-threaded systems in Section 4.4. In Section 4.5 we present a technique to reduce the model checking state-space by only model checking interleavings until the point that they would blocked at runtime by the resource semantics. We conclude in Section 4.6.

4.2 Related Work

In [32] the authors present a type system for ensuring that resources are accessed according to predefined patterns. This is a formulaically elegant and general analysis which could be used in many diverse situations. Hypothetically the analysis in [32], slightly modified to ensure complementary actions between roles, could be used as a session typing system. The analysis in [32] could also probably be used to control and validate shared memory access. This work, however, only considers single-threaded functional programs, and only permits specifying policies on single resources.

There is significant work on validating correct use of resources with respect to local automata policies [3, 5–7]. This work permits policies to be specified with respects to multiple resources, which can be both static or dynamic (created at runtime). In [5] the authors present the concept of local policies, which only take account of specific actions on specific resources - all other actions are ignored. This approach allows a policy designer to focus on the property they are interested rather than the global state of available resources, as previous approaches have done [8]. In [3, 6, 7] the authors present a methodology for efficiently model checking the policies against the actions of a program. Of particular note is their approach of transforming resource usages into Basic Process Algebras and using a weakened form of model checking to make their approach complete as well as sound. The authors posit that their approach could be simply extended to multi-threaded programs by transforming into Basic Parallel Processes [18] rather than BPAs [7]. Recent work in equivalence decidability [30], however, shows that trace equivalence, on which their approach is based, is undecidable for BPPs. We instead look at representing the behaviour of multi-threaded programs as the behaviour of single-threaded programs, which can be transformed to BPAs as in [7].

Extensive research exists on usage policies and their enforcement mechanisms which are not relevant here. For a fuller review of the area see [3].

4.3 Access Policies for Single-Threaded Programs

We present existing work on access policies from single-threaded programs. This is a summary of [3, 5–7] with very minor modifications to aid our approach. Access policies act on *events* $\alpha(T)$, where α is the event name (such as 'lock' or 'read') and T denotes the type of the data used by the event. We sometimes use a label L to denote an event. We make use of a basic calculus for usages, which can be computed for actual program code using a type and effect system [40]. We define this calculus in Figure 4.1.

Usages are history expressions. Actions, recursions, sequencing, and internal and external choice are straightforward. Name binding $\nu x.U$ is a binder on the occurrences of the

$$
\begin{array}{lcl}
U & ::= & \epsilon \quad \mid \quad U + U \\
 & \mid & \alpha(T) \quad \mid \quad \nu x.U \\
 & \mid & \underline{\mathsf{h}} \quad \mid \quad \mu\underline{\mathsf{h}}.U \\
 & \mid & U;U \quad \mid \quad \varphi[U]
\end{array}
\qquad
\begin{array}{lcl}
T & ::= & \textsc{Bool} \\
 & \mid & \textsc{Int} \\
 & \mid & \mathsf{Res}\,x \\
 & \mid & \mathsf{Res}\,r \\
 & \mid & T \to T
\end{array}
$$

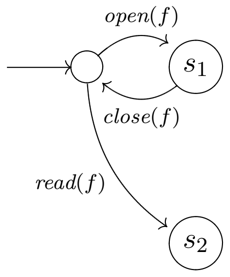
FIGURE 4.1: Usage and Type Grammars

name x in U in order to keep track of the binding between x and a freshly created resource. We use $\varphi[U]$ as a sandboxing construct which denotes that the policy φ must be respected during the execution of U. The $\varphi[U]$ construct is syntactic sugar for $[_\varphi; U;]_\varphi$. Note that when checking a policy we examine the trace before the start of the policy and require that it, combined with the usage inside the policy, to be valid with respects to the policy. This is called a history based approach.

Types include special nominal types: $\mathsf{Res}\,r$ denotes a resource r, and $\mathsf{Res}\,x$ denotes a dynamic resource variable x which has yet to be instantiated; we use nominal types to prevent aliasing. As events which act on multiple resources can be encoded using events which act on single resources [3] we abuse notation and use polyadic events such as 'write($\mathsf{Res}\,z$, $\textsc{Int}$)'. A *trace* is a finite sequence of events, typically denoted by η.

We define usage policies, which define some regular property, e.g. properly opening and closing a file:

Example 4.1.



where s_1 and s_2 are sink states. We assume that a sink state is one in which it would be erroneous to end; it is acceptable to open a file, read it, then close it, but failing to close the file is incorrect. Intuitively we would include a *read* self loop on s_1 to indicate that reading is permissible in this state, along with self loops for all actions on s_2; we omit these as self loops will be automatically added later.

Formally, a usage policy $\varphi(x)$ is a 5-tuple $\langle B, Q, q_0, F, E\rangle$, where B is the input alphabet (a set of events) which includes $\mathsf{Res}\,x$ and $\overline{\mathsf{Res}\,x}$, Q is a finite set of states, q_0 is the starting state, F is the set of final (offending) states, and E is a finite set of edges. Finally, x is the variable in which the usage policy is parametric - it can be instantiated with any resource which is created at runtime (any *dynamic resource*). Edges in a usage policy can be of three kinds: either $\alpha(T)$ for a static type T (which includes static resources), or $\alpha(\mathsf{Res}\,x)$ for the parametric variable resource, or $\alpha(\overline{\mathsf{Res}\,x})$ for any other dynamic resource besides $\mathsf{Res}\,x$. We also make use of the syntactic sugar $\mathsf{Res}\,*$ to denote any resource and $*$ to denote any type.

Given a resource $\mathsf{Res}\,r$, a set of resources R, and a set of event names Events, a usage policy $\varphi(x)$ is instantiated into a policy automaton $A_{\varphi(r,R)}$ by binding x to the resource $\mathsf{Res}\,r$ and making $\overline{\mathsf{Res}\,x}$ range over $\{\mathsf{Res}\,r' | r' \in R \setminus r\}$. Additionally we add self loops for all events in the alphabet which we're not interested in, formally:

Definition 4.1. Usage Policies

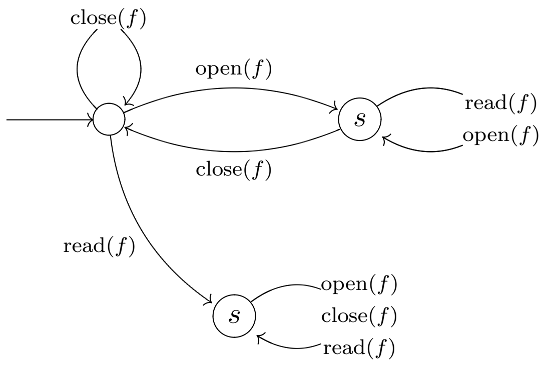
$$A_{\varphi(r,R)} = \langle B', Q, q_0, F, E'' \rangle$$

where

$$\begin{aligned}
B' &= \{\alpha(T) | \alpha \in \text{Events} \wedge T \in R\} && \\
E' &= \{q \xrightarrow{\alpha(\mathsf{Res}\,r)} q' | q \xrightarrow{\alpha(\mathsf{Res}\,x)} q' \in E\} \cup && \text{instantiation of } x \\
&\quad \textstyle\bigcup_{r' \in (R \cup \{?\}) \setminus r} \{q \xrightarrow{\alpha(\mathsf{Res}\,r')} q' | q \xrightarrow{\alpha(\overline{\mathsf{Res}\,x})} q' \in E\} \cup && \text{instantiation of } \overline{x} \\
&\quad \{q \xrightarrow{\alpha(T)} q' | q \xrightarrow{\alpha(T)} q' \in E \wedge T \neq \mathsf{Res}\,x, \overline{\mathsf{Res}\,x}\} && \text{static resources} \\
E'' &= E' \cup \{q \xrightarrow{\alpha(T)} q' | \nexists q \xrightarrow{\alpha(T)} q' \in E'\} && \text{self loops}
\end{aligned}$$

Unknown resources [7] are wild cards allow slightly more complex properties to be checked. Our formulation here leaves out unknown resources for the sake of simplicity, but is capable of dealing with them in a way which is consistent with the underlying model checking in [7] which we make use of.

In a world where the only actions were *open*, *read*, and *write*, and the only resource is f, this closure would result in a FSA similar to:



We define that when a trace respects all the relevant usage policies it is *valid*. In order to define this formally we include several auxiliary concepts. We consider making a trace η frameless $\mathsf{unbox}(\eta)$, which means removing all policy frames $[_\varphi,]_\varphi$ from η. We also consider active policies within a trace; conceptually active policies are those which we are currently enforcing. An active policy φ is one for which there is an opening

$$\frac{}{\alpha(T),R\xrightarrow{\alpha(T)}\epsilon,R}\ (\text{URAction}) \qquad \frac{r\in\text{Res}_d\setminus R}{\nu x.U,R\xrightarrow{\epsilon}U[r/x],R\cup\{r\}}\ (\text{URNewName})$$

$$\frac{}{\epsilon;U,R\xrightarrow{\epsilon}U,R}\ (\text{URSeqEmpt}) \qquad \frac{U_1,R\xrightarrow{\eta}U_1',R'}{U_1;U_2,R\xrightarrow{\eta}U_1';U_2,R'}\ (\text{URSeq})$$

$$\frac{i\in 1,2}{U_1+U_2,R\xrightarrow{\epsilon}U_i,R'}\ (\text{URIntChoice}) \qquad \frac{i\in 1,2}{U_1\&U_2,R\xrightarrow{\epsilon}U_i,R'}\ (\text{URExtChoice})$$

$$\frac{}{\mu\underline{\mathsf{h}}.U,R\xrightarrow{\epsilon}U[\mu\underline{\mathsf{h}}.U/\underline{\mathsf{h}}],R}\ (\text{URRecDef})$$

FIGURE 4.2: Usage Reduction Semantics

frame $[_\varphi$ for that policy within the trace, but there is no corresponding closing frame $]_\varphi$. The multiset act denotes the active policies of a trace. A trace η' is a sub trace of η if $\eta = \eta';\ \eta''$. We then define a trace as valid if at no point in the trace do we contrevene any policy. Formally:

Definition 4.2. Frameless Traces, Active Policies, and Valid Trace Definitions

$$\mathsf{unbox}(\eta) \overset{\text{def}}{=} \begin{cases} \eta & \eta = \alpha(T) \\ \epsilon & \eta = [_\varphi \vee \eta =]_\varphi \\ \mathsf{unbox}(\eta_1); \mathsf{unbox}(\eta_2) & \eta = \eta_1;\ \eta_2 \end{cases}$$

$$\mathsf{act}(\eta) \overset{\text{def}}{=} \begin{cases} \emptyset & \eta = \epsilon \\ \mathsf{act}(\eta') & \eta = \eta'; \alpha(T) \\ \mathsf{act}(\eta') \cup \{\varphi\} & \eta = \eta'; [_\varphi \\ \mathsf{act}(\eta') \setminus \{\varphi\} & \eta = \eta';]_\varphi \end{cases} \qquad \frac{}{\vDash \epsilon} \qquad \frac{\vDash \eta \quad \forall\varphi \in \mathsf{act}(\eta; L).\ \mathsf{unbox}(\eta; \mathsf{L}) \vDash \varphi}{\vDash \eta; L}$$

where $\eta \vDash \varphi$ when η is *not* included in the language defined by the automaton defined by φ. Conceptually a trace is valid when all sub traces respect the active policies within those sub traces.

The traces which can be obtained from a usage U are defined to be those which can be obtained by reducing the usage $U, R \xrightarrow{\eta}{}^* U', R'$ using the rules in Figure 4.2. A usage U is valid when, for all $U', R, R', \eta, (U, R \xrightarrow{\eta} U', R')$ we have $\vDash \eta$.

In order to check whether a trace respects (is valid with respects to) a policy we would have to instantiate infinitely many automaton, each of which would have infinitely many edges (though a finite number of states). However, using an approach demonstrated in [7] this can be decided through a finite set of finite state automata. The approach in [7] contains additional technicalities; as we can plug the results of our contribution directly into their infrastructure we do not go into these technicalities in detail.

4.4 Access Policies for Multi-Threaded Programs

Extending usages for single-threaded programs (Definition 4.1) to concurrent usages for multi-threaded programs (Definition 4.3) is straightforward. We permit resource creation to span over concurrent usages, and permit concurrent usages to be put in parallel with each other.

Definition 4.3. Parallel Usages Grammar

$$\begin{array}{lcl} P & ::= & U \\ & | & \nu x.P \\ & | & P \| P \end{array}$$

In [7] the authors posit that their approach (outlined in Section 4.3) could be simply extended to multi-threaded programs by using concurrent usages and transforming them into Basic Parallel Processes rather than BPAs. Recent work in equivalence decidability [30], however, shows that trace equivalence, on which their approach is based, is undecidable for BPPs.

Instead, our approach is to look at representing the behaviour of multi-threaded programs as the behaviour of single-threaded programs. If we can transform a concurrent usage into a single-threaded usage which is trace equivalent to the concurrent usage, then we can model check this single-threaded usage as in [7].

4.4.1 Projecting Multi-Threaded Usages onto Single-Threaded Usages

Projecting concurrent usages onto single-threaded usages is straightforward if the concurrent usage is either a single-threaded usage or a dynamic resource creation construct. In order to explain how we project two usages in parallel to a trace equivalent single-threaded usage we illustrate with the following example. Consider the concurrent usage $\alpha_1; \alpha_2 \| \alpha_3; \alpha_4$. When identifying the possible traces which can occur from this usage, we can either choose for action α_1 to occur first, or for action α_3 to go first. In order to reflect both possible traces we must insert a non-deterministic choice between the two. Conceptually, the result of running trace on this example would be:

$$\mathsf{trace}(\alpha_1; \alpha_2 \| \alpha_3; \alpha_4) = \alpha_1; \mathsf{trace}(\alpha_2 \| \alpha_3; \alpha_4) + \alpha_3; \mathsf{trace}(\alpha_1; \alpha_2 \| \alpha_4)$$

Effect systems often use constraints in order to generate the effects [40]. When detecting and modelling recursion explicitly we obtain constraints such as:

$$U = \alpha(T); U$$

$$\mathsf{Exp}(P,\Gamma) \stackrel{\text{def}}{=} \begin{cases} U & P = U \wedge \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P \\ \nu x.\mathsf{Exp}(P',\Gamma) & P = \nu x.P' \wedge \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P \\ \mathsf{next}(\mathsf{Exp}(P_1,\Gamma),\epsilon,P_2,\Gamma)+ & P = P_1 \| P_2 \wedge \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P \\ \mathsf{next}(\mathsf{Exp}(P_2,\Gamma),\epsilon,P_1,\Gamma) & \\ \underline{\mathsf{h}} & \exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P \end{cases}$$

$$\mathsf{next}(U,U',P,\Gamma) \stackrel{\text{def}}{=} \begin{cases} \mathsf{Exp}(P\|U',\Gamma) & U = \epsilon \\ \alpha(T);\mathsf{Exp}(P\|U',\Gamma) & U = \alpha(T) \\ \mathsf{next}(U,U_2;U',P,\Gamma) & U = U_1;U_2 \\ \mathsf{next}(U_1,U',P,\Gamma)+ & U = U_1 + U_2 \\ \mathsf{next}(U_2,U',P,\Gamma) & \\ \nu x.\mathsf{next}(U'',U',P,\Gamma) & U = \nu x.U'' \\ \mu\underline{\mathsf{h}}'.\mathsf{next}(U''',U',P,\Gamma') & U = \mu\underline{\mathsf{h}}.U'' \wedge \\ & \nexists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = U;U'\|P \\ & \underline{\mathsf{h}}'' \text{ free in } \Gamma \wedge \\ & U''' = U''[U/\underline{\mathsf{h}}] \wedge \\ & \Gamma' = \Gamma[\underline{\mathsf{h}}'' \mapsto P\|U;U'] \\ \underline{\mathsf{h}} & U = \mu\underline{\mathsf{h}}.U'' \wedge \\ & \exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = U;U'\|P \end{cases}$$

FIGURE 4.3: Trace Projection of Multi-Threaded Usages

This is the effect of a recursive function such as:

$$\lambda_f x.\mathsf{acc}^{\alpha(x)};\ f\,x$$

where $\mathsf{acc}^{\alpha(x)}$ is a construct which performs action $\alpha(x)$. When we obtain such a constraint we resolve it using a recursive definition such as:

$$U = \mu\underline{\mathsf{h}}.(\alpha(T);\underline{\mathsf{h}})$$

We use a similar technique in our trace definition. The approach is more complicated, however, as we have to deal with recursion under the parallel operator. Conceptually what we do is the following:

$$\begin{aligned}
&\mathsf{Exp}(\mu\underline{\mathsf{h}}.(L_1;\underline{\mathsf{h}})\|\mu\underline{\mathsf{h}}'.(L_2;\underline{\mathsf{h}}')) \\
&\quad = \mathsf{Exp}(P_1\|P_2) \\
&\quad = \mathsf{next}(\mathsf{Exp}(P_1),\epsilon,P_2) + \mathsf{next}(\mathsf{Exp}(P_2),\epsilon,P_1) \\
&\quad = \mathsf{next}(P_1,\epsilon,P_2) + \mathsf{next}(P_2,\epsilon,P_1) \\
&\quad = \mathsf{next}(L_1;P_1,\epsilon,P_2) + \mathsf{next}(L_2;P_2,\epsilon,P_1) \\
&\quad = \mathsf{next}(L_1,P_1,P_2) + \mathsf{next}(L_2,P_2,P_1) \\
&\quad = L_1;\mathsf{Exp}(P_1\|P_2) + L_2;\mathsf{Exp}(P_2\|P_1)
\end{aligned}$$

This should end up with a trace equivalent projection such as $\mu\underline{\mathsf{h}}.(L_1+L_2);\underline{\mathsf{h}}$. In order to be able to constructively generate such projections we maintain a record of what contexts we have seen a recursive definition in, using Γ and Σ; a context consists of the effect sequentially following the definition within that single-threaded effect, and the concurrent effects which are in parallel. We separate these definitions in order to aid the proof that this technique is well defined and finite. For more details see Section 4.4.3.

We formally define the projection functions in Figure 4.3. In the $\mathsf{Exp}(P, \Gamma)$ function we use Γ (referred to as the created recursive definitions) to keep track of the contexts in which we have seen recursive definitions, where a context is the usage sequentially following the recursive definition together with the concurrent usage it is in parallel with.

4.4.2 Possible Traces of Parallel Usages

In order to check trace equivalence we need to know what possible traces a usage or a concurrent usage can perform. This is straightforward for single-threaded usages:

Definition 4.4. Interleavings of Single Threaded Usages

$$\mathsf{Traces}(U) \overset{\text{def}}{=} \{\eta | U, \emptyset \xrightarrow{\eta} U', R'\}$$

using the reduction semantics defined in Figure 4.2. In order to extend this approach to concurrent usages we must extend the reduction semantics to include the following rules:

$$\frac{r \in \mathrm{Res}_d \backslash R}{\nu x.P, R \xrightarrow{\epsilon} P[r/x], R \cup \{r\}} \;(\text{URParNewName})$$

$$\frac{P_1, R \xrightarrow{\eta} P_1', R'}{P_1 \| P_2, R \xrightarrow{\eta} P_1' \| P_2, R'} \;(\text{URParOne}) \qquad \frac{P_2, R \xrightarrow{\eta} P_2', R'}{P_1 \| P_2, R \xrightarrow{\eta} P_1 \| P_2', R'} \;(\text{URParTwo})$$

Then we can define:

Definition 4.5. Interleavings of Concurrent Usages

$$\mathsf{Traces}(P) \overset{\text{def}}{=} \{\eta | P, \emptyset \xrightarrow{\eta} P', R'\}$$

4.4.3 Trace Equivalence of Multi-Threaded Usages and Their Projections

In order to show trace equivalence we must first show that the trace function is finite and well defined. We do this by transforming the function into a tableau form. We include some representative tableau rules in Figure 4.4. We make use of standard equivalence rules for parallel processes.

Theorem 4.6. *Each* trace *tableau is finite.*

$$\frac{\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv U}{U \vdash \mathsf{Exp}(U, \Delta, \Gamma)}\ (\text{:TUsage}) \qquad \frac{\exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P}{\underline{\mathsf{h}} \vdash \mathsf{Exp}(P, \Delta, \Gamma)}\ (\text{:TParTerm})$$

$$\frac{\begin{array}{c}\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P_1 \| P_2 \qquad U_1 \vdash \mathsf{Exp}(P_1, \Delta, \Gamma) \qquad U_2 \vdash \mathsf{Exp}(P_2, \Delta, \Gamma) \\ U_1' \vdash \mathsf{next}(U_1, \emptyset, P_2, \Delta, \Gamma, \emptyset) \qquad U_2' \vdash \mathsf{next}(U_2, \emptyset, P_1, \Delta, \Gamma, \emptyset)\end{array}}{U_1' + U_2' \vdash \mathsf{Exp}(P_1 \| P_2, \Delta, \Gamma)}\ (\text{:TPar})$$

$$\frac{U \vdash \mathsf{Exp}(P \| U', \Delta, (\Gamma, \Sigma))}{\alpha(T); U \vdash \mathsf{next}(\alpha(T), U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TAction})$$

$$\frac{U \vdash \mathsf{next}(U'', U', P, \Delta[\underline{\mathsf{h}} \mapsto \mu \underline{\mathsf{h}}.U''], \Gamma, \Sigma[\underline{\mathsf{h}}' \mapsto P \| \mu \underline{\mathsf{h}}.U''; U']) \qquad \underline{\mathsf{h}}' \text{ free in } \Delta, \Gamma, \Sigma}{\mu \underline{\mathsf{h}}'.U \vdash \mathsf{next}(\mu \underline{\mathsf{h}}.U'', U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TRecDef})$$

Figure 4.4: Representative Tableau Rules

We make use of a proof method based on [47] which shows that there cannot be an infinite path through the tableau. An infinite path could only occur from the recursive definitions, but as each time we encounter a recursive definition in parallel with a concurrent usage we record it, in association with a fresh recursive variable (e.g. in TRecDef, Figure 4.4). As this is a finite system we will eventually encounter the same recursive definition in parallel with the same concurrent usage, and we can terminate returning the previously created recursive variable.

Having proved well definedness and finiteness of the tableau methods we can then prove trace equivalence of the projection of a concurrent usage onto a single-threaded usage.

Theorem 4.7. $\mathsf{Traces}(P, \Gamma) = \mathsf{Traces}(\mathsf{Exp}(P, \Gamma), \Gamma)$

We prove this by simulataneous lexicographic induction over (Γ, P) for int and trace, where:

$$(\Gamma, P) < (\Gamma', P') \overset{\text{def}}{=} (\Gamma = \Gamma', \Gamma'') \vee \Gamma = \Gamma' \wedge P < P'$$

This induction is straightforward apart from in the case where $P = P_1 \| P_2$ and $\mathsf{Exp}(P_1, \Gamma) \equiv \mu \underline{\mathsf{h}}.U'$. We extend Γ when we come across a recursive definition in a new context. If we have not seen the definition in this context we increase the context set Γ and make use of the fact that trace with a larger Γ are lexographically smaller. If we have seen this definition in this context before we simply return the recursive variable that represents the effect of the definition in this context. As we are working with a finite state system the number of possible contexts is bounded, and hence we can perform induction over increasing Γ.

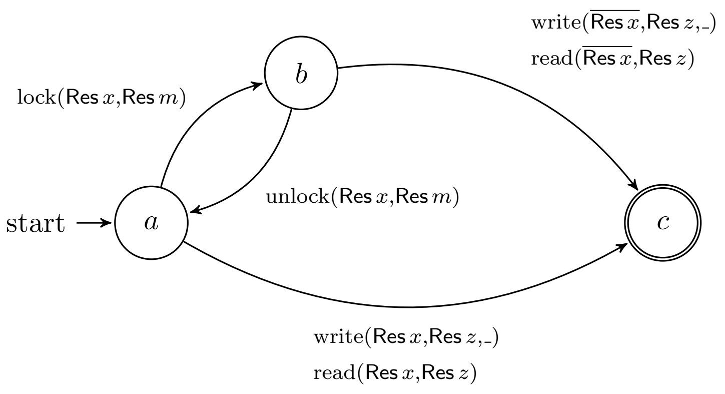

write($\overline{\mathsf{Res}\,x}$,$\mathsf{Res}\,z$,_)
read($\overline{\mathsf{Res}\,x}$,$\mathsf{Res}\,z$)
lock($\mathsf{Res}\,x$,$\mathsf{Res}\,m$)
b
unlock($\mathsf{Res}\,x$,$\mathsf{Res}\,m$)
start
a
c
write($\mathsf{Res}\,x$,$\mathsf{Res}\,z$,_)
read($\mathsf{Res}\,x$,$\mathsf{Res}\,z$)

Figure 4.5: Mutex Policy Automata

4.5 Model Checking Access Policies

Once we have converted a concurrent usage into a trace equivalent single-threaded usage we can model check it using the approach from [7]. If we naively use this approach (Section 4.5.1) the complexity is exponential in the number of parallel branches [34]. We can reduce the possible interleavings which we have to model check by taking account of the dynamic semantics of the accompanying resources (Section 4.5.2). That approach, however, requires us to evaluate the resources along with the model checking. As an alternative we present the novel approach of permitting access policies to ignore certain interleavings, as long as they simulate the resource semantics (Section 4.5.3). This ensures that model checking using access policies to decide which traces to ignore conservatively approximates model checking using the resources and their semantics to decide which traces to ignore.

4.5.1 Naive Model Checking

Consider the following concurrent usage:

Example 4.2.

$$\begin{aligned}&\mathit{lock}(\mathsf{Res}\,i, \mathsf{Res}\,m); \mathit{write}(\mathsf{Res}\,i, \mathsf{Res}\,z, \textsc{Int}); \mathit{unlock}(\mathsf{Res}\,i, \mathsf{Res}\,m)\|\\ &\mathit{lock}(\mathsf{Res}\,j, \mathsf{Res}\,m); \mathit{write}(\mathsf{Res}\,j, \mathsf{Res}\,z, \textsc{Int}); \mathit{unlock}(\mathsf{Res}\,j, \mathsf{Res}\,m)\\ &\quad = L_1; L_2; L_3\|L_4; L_5; L_6\end{aligned}$$

where i, j are thread identifiers to indicate which thread is performing the action, m is a mutex name, and z is a shared variable name. We may wish to model check this usage against a standard mutex policy, as defined in Figure 4.5. where s is the sink, and we would instantiate $\mathsf{Res}\,x$ as $\mathsf{Res}\,i$ and $\mathsf{Res}\,j$.

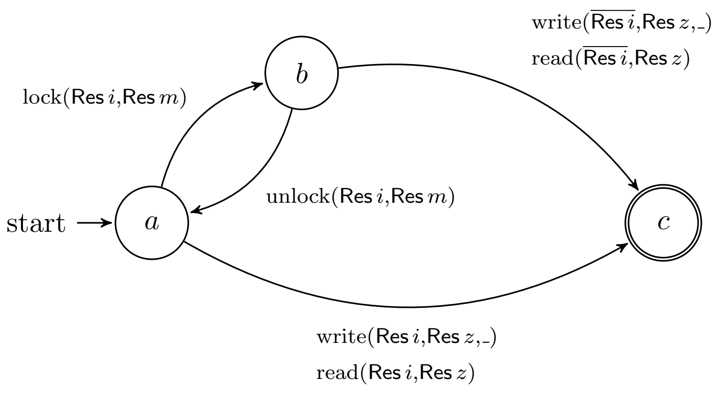

write($\overline{\mathsf{Res}\,i}$,$\mathsf{Res}\,z$,_)
read($\overline{\mathsf{Res}\,i}$,$\mathsf{Res}\,z$)
lock($\mathsf{Res}\,i$,$\mathsf{Res}\,m$)
b
unlock($\mathsf{Res}\,i$,$\mathsf{Res}\,m$)
start
a
c
write($\mathsf{Res}\,i$,$\mathsf{Res}\,z$,_)
read($\mathsf{Res}\,i$,$\mathsf{Res}\,z$)

FIGURE 4.6: Instantiated Mutex Policy Automata

If we naively interleave the concurrent usages, we get 16 possible traces:

$$\begin{array}{l} L_1; L_2; L_3; L_4; L_5; L_6 \\ L_4; L_1; L_2; L_3; L_4; L_5 \\ \ldots \\ L_4; L_1; L_5; L_6; L_2; L_3 \\ L_4; L_5; L_6; L_1; L_2; L_3 \end{array}$$

In a naive model checking scenario we would check to ensure that each possible interleaving does not end in the sink state. Obviously an interleaving such as:

$$\text{lock}(\mathsf{Res}\,i, \mathsf{Res}\,m); L_4; \text{write}(\mathsf{Res}\,j, \mathsf{Res}\,z, \textsc{Int}); L_6; L_2; L_3$$

would invalidate the automata above (Figure 4.5) instantiated with identifier i (Figure 4.6). Hence the usage in Example 4.2 would be considered unsafe with respects to the automata in Example 4.5, despite the fact that the usage is clearly a safe use of the shared variable and its mutex, under standard semantics for locking and unlocking. In order to get a more fine-grained analysis we must take account of these semantics.

4.5.2 Model Checking Using Resources

Whilst there are 16 possible naive interleavings of Example 4.2, only two of them can actually occur in a system with standard semantics for locking and unlocking of mutexes. These interleavings are as follows:

$$L_1; L_2; L_3; L_4; L_5; L_6 \qquad\qquad L_4; L_5; L_6; L_1; L_2; L_3$$

These represent either the first thread goes first and runs to completion and is followed by the second thread, or vice-versa.

Conceptually we need only model check traces of a usage until it becomes blocked; any infractions of the policy after the point which a trace becomes stuck will never occur in practice. In order to determine when a trace becomes blocked we need an abstract model of the resources and their semantics to run the trace on. For our running example the resources and the semantics of those resources are defined in Figure 4.7, where $\mathsf{Res}\,m{\mapsto}\mathsf{locked}(\mathsf{Res}\,i)$ denotes that mutext m is locked by thread i and $\mathsf{Res}\,z{\mapsto}T$ denotes that variable z contains a value of type T. The semantics denotes how an performing an action L on the resources modifies those resources.

Formally the set of traces of the longest possible subtraces of the interleavings before blocking occurs, for resources σ and resource semantics $\sigma \rightarrow \sigma'$, is defined as:

Definition 4.8. $\mathsf{O}^{\sigma}(U) \stackrel{\text{def}}{=} \{\eta' | \eta \in U \wedge \eta' \text{ is the largest prefix of } \eta \text{ s.t. } \sigma_{init} \xrightarrow{\eta'}{}^{*}\}$

where σ_{init} represents the starting state of resources, $\sigma_{init} \xrightarrow{\eta'}{}^{*} \sigma'$ is the transitive closure of the semantics, and $\eta \in U$ if

$$\exists R, R', U'.(U, R \xrightarrow{\eta} U', R')$$

In our example the starting state of resources may be a mapping of shared variables to default value types and mutexes to unlocked.

Whilst in this small example we can easily reduce the traces using the resource semantics to find the largest prefix, in a larger, real-world system the cost in terms of time or space to model check using the resources may be unreasonable.

4.5.3 Model Checking Using Simulating Automata

In order to reduce the computational requirements of model checking access policies we want to find a way to eliminate or reduce the usage of the resource semantics in the model checking, as running this in parallel with the model checking adds continuous

$$\begin{array}{rl} \sigma ::= & \mathsf{Res}\,m{\mapsto}\mathtt{unlocked} \\ | & \mathsf{Res}\,m{\mapsto}\mathsf{locked}(\mathsf{Res}\,i) \\ | & \mathsf{Res}\,z{\mapsto}T \\ | & \sigma{,}\sigma \end{array} \qquad \frac{}{\sigma \xrightarrow{read(\mathsf{Res}\,z)} \sigma} \qquad \frac{\sigma' = \sigma[\mathsf{Res}\,z \mapsto T]}{\sigma \xrightarrow{write(\mathsf{Res}\,z,T)} \sigma'}$$

$$\frac{\sigma(\mathsf{Res}\,m) = \mathtt{unlocked} \quad \sigma' = \sigma[\mathsf{Res}\,m \mapsto \mathsf{locked}(\mathsf{Res}\,i)]}{\sigma \xrightarrow{lock(\mathsf{Res}\,i,\mathsf{Res}\,m)} \sigma'}$$

$$\frac{\sigma(\mathsf{Res}\,m) = \mathsf{locked}(\mathsf{Res}\,i) \quad \sigma' = \sigma[\mathsf{Res}\,m \mapsto \mathtt{unlocked}]}{\sigma \xrightarrow{unlock(\mathsf{Res}\,i,\mathsf{Res}\,m)} \sigma'}$$

FIGURE 4.7: Resource Definitions and Semantics

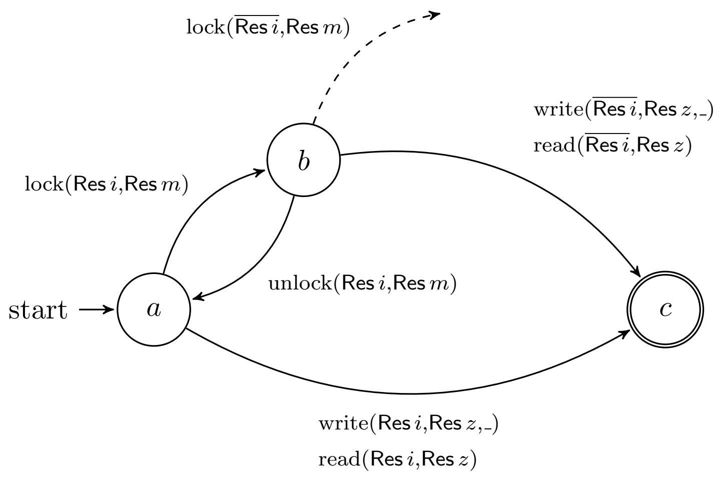

lock($\overline{\mathsf{Res}\,i}$,$\mathsf{Res}\,m$)
write($\overline{\mathsf{Res}\,i}$,$\mathsf{Res}\,z$,_)
read($\overline{\mathsf{Res}\,i}$,$\mathsf{Res}\,z$)
lock($\mathsf{Res}\,i$,$\mathsf{Res}\,m$)
b
unlock($\mathsf{Res}\,i$,$\mathsf{Res}\,m$)
start
a
c
write($\mathsf{Res}\,i$,$\mathsf{Res}\,z$,_)
read($\mathsf{Res}\,i$,$\mathsf{Res}\,z$)

FIGURE 4.8: Refusal Policy for Mutex Usage

runtime overhead. In order to do this we introduce a modified form of access policies called *refusal policies* (these are defined by the programmer similarly to usage policies in [7]). Refusal policies include dotted arrows (a *refusal arrow*) which indicates that a transition which we deem impossible to occur from the arrow's starting state. This means that, when we generate the usage automaton from the model, we do not add self loops for that action on that state. An example refusal policy is shown in Figure 4.8. Formally, a refusal usage policy $\varphi(x)$ is a 6-tuple $\langle B, Q, q_0, F, E, D\rangle$, where B, Q, q_0, F, and E are as in Definition 4.1. The placeholder D denotes the refusal edges $q \overset{L}{\not\to}$, the transitions which are deemed to be impossible to occur at runtime for a given state. The refusal policy in Figure 4.8 would have $D = \{\, b \xrightarrow{\text{lock}(\overline{\mathsf{Res}\,i},\mathsf{Res}\,m)}\!\not\quad\quad\quad\quad\quad\quad\,\}$.

Given a refusal usage policy $\langle B, Q, q_0, F, E, D\rangle$ we can instantiate it into an access policy automaton, which has the same form as before:

Definition 4.9. Refusal Usage Policies

$$A_{\varphi(r,R)} = \langle B', Q, q_0, F, E''\rangle$$

where

$$\begin{aligned}
B' &= \{\alpha(T) | \alpha \in \text{Events} \wedge T \in R\} && \\
E' &= \{q \xrightarrow{\alpha(\mathsf{Res}\,r)} q' | q \xrightarrow{\alpha(\mathsf{Res}\,x)} q' \in E\} \cup && \text{instantiation of } x \\
&\quad \textstyle\bigcup_{r' \in (R \cup \{?\}) \setminus r} \{q \xrightarrow{\alpha(\mathsf{Res}\,r')} q' | q \xrightarrow{\alpha(\overline{\mathsf{Res}\,x})} q' \in E\} \cup && \text{instantiation of } \overline{x} \\
&\quad \{q \xrightarrow{\alpha(T)} q' | q \xrightarrow{\alpha(T)} q' \in E \wedge T \neq \mathsf{Res}\,x, \overline{\mathsf{Res}\,x}\} && \text{static resources} \\
E'' &= E' \cup \{q \xrightarrow{\alpha(T)} q' | \nexists q \xrightarrow{\alpha(T)} q' \in E' \wedge \nexists\, q \overset{L}{\not\to} \in D\} && \text{self loops minus} \\
& && \text{blocked actions}
\end{aligned}$$

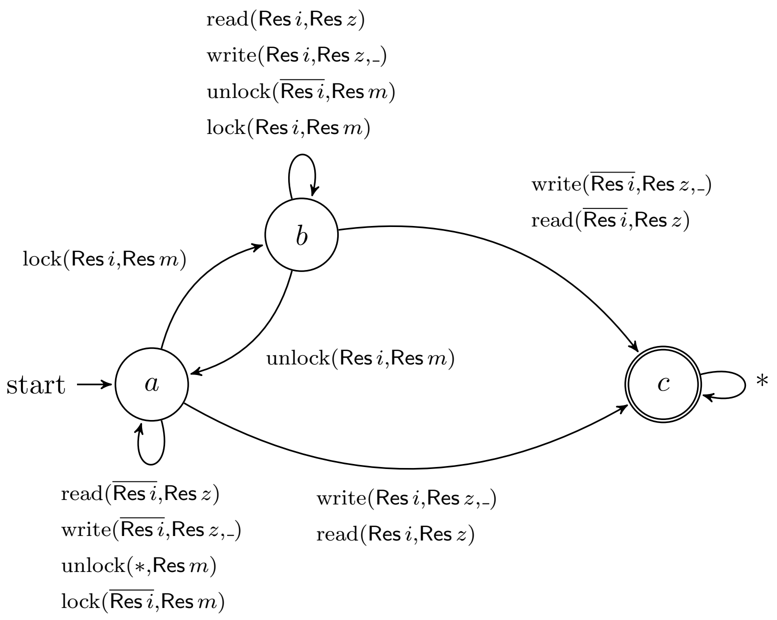

read(Res i,Res z)
write(Res i,Res z,_)
unlock($\overline{\text{Res}\,i}$,Res m)
lock(Res i,Res m)
write($\overline{\text{Res}\,i}$,Res z,_)
read($\overline{\text{Res}\,i}$,Res z)
lock(Res i,Res m)
b
unlock(Res i,Res m)
start
a
c
*
read($\overline{\text{Res}\,i}$,Res z)
write($\overline{\text{Res}\,i}$,Res z,_)
unlock($*$,Res m)
lock($\overline{\text{Res}\,i}$,Res m)
write(Res i,Res z,_)
read(Res i,Res z)

FIGURE 4.9: Completed Refusal Policy for Mutex Usage

Note the difference in the definition of E'': we do not add a self loop if the actions is considered impossible in that state. If we perform this instantiation on the refusal policy in Figure 4.8 we obtain the automata in Figure 4.9. Note that there is no self loop for lock($\overline{\text{Res}\,i}$, Res m, _) from state b. If, when checking a potential trace, we reach a state where there is no transition for the next action in the trace then we assume that the action is impossible and that the remainder of the trace is inaccessible due to a blocked action. In such a case we don't need to check the remainder of the trace, only to see if the blocked state is a sink. Conceptually, we only model check a trace until the automaton blocks; formally, the set of traces we will check is:

Definition 4.10. $\mathsf{O}^{\mathsf{A}_{\varphi(\mathsf{r},\mathsf{R}(\mathsf{U}))}}(U) \stackrel{\text{def}}{=} \{\eta' | \eta \in U \wedge \eta' \text{ is largest prefix of } \eta \text{ s.t. } q_0 \xrightarrow{\eta'}{}^{*}\}$

where $A_{\varphi(r,\mathsf{R}(U))} = \langle \_, \_, q_0, \_, E\rangle$ and $q_0 \xrightarrow{\eta'}{}^{*}$ is the transitive closure of the reduction relation E.

In order for model checking this set of traces to conservatively approximate model checking using the resource semantics we require that each time the policy automaton is blocked for a certain trace the resource semantics would also be blocked; conceptually, any trace that we don't check due to blocking in the automaton must be a trace which we wouldn't have checked due to blocking in the resource semantics. We can ensure this property holds by requiring each automaton $A_{\varphi(r,\mathsf{R}(U))}$ to simulate the resource semantics. We can generate a simulation relation using standard techniques such as [19]. Given such a simulation relation, we can show that:

Lemma 4.11. *If $A_{\varphi(r,\mathsf{R}(U))}$ simulates σ with relation R then*

$$\forall (q, \sigma) \in R.\, q \overset{\not\iota}{\not\to} \ \Rightarrow \ \sigma \overset{\not\iota}{\not\to}$$

This is trivially derivable from the definition of simulation. Given this property then we can prove the following:

Theorem 4.12. *If all traces defined by $\mathsf{O}^{\mathsf{A}_{\varphi(\mathsf{r},\mathsf{R}(\mathsf{U}))}}(U)$ are valid for $A_{\varphi(r,\mathsf{R}(U))}$ then all traces defined by $\mathsf{O}^{\sigma}(U)$ are valid for $A_{\varphi(r,\mathsf{R}(U))}$.*

Hence, assuming that our automaton simulates the resource semantics, we model check using the automaton rather than using the resource semantics. The cost of this approach is generating the simulation relation. Given that the policy automaton is likely to change far less frequently than the behaviour of a program during development, this should significantly reduce computation costs.

4.6 Conclusions

We demonstrate how to represent a concurrent usage using a trace equivalent single-threaded usage. We use this representation to permit us to model check usages from multi-threaded code on existing machinery [7] for model checking single-threaded code. We extend the existing machinery to an analysis which disregards blocked traces. As this analysis is more finely-grained we can recognise correctness where before we would have returned a false positive with regards to errors (Example 4.2). This approach also reduces the computational complexity of model checking as we only need to model check using the policy automaton rather than using the automaton and the resource semantics.

Chapter 5

Dynamic Update

5.1 Introduction

There are various situations where the code of a program may be modified at run-time. Operating systems may permit various extensions (normally drivers) to be installed to better manage resources, for example per application thread schedulers or an alternate disk access and caching for databases [9]. Commercial products may permit plugins to customise them, for example in web browsers which can incorporate functionality from control of music players to highlighting of web pages. Self-modifying code modifies run-time code for many reasons, most notably in testing to add or remove debugging code without performing conditional jumps and in just-in-time compilation to optimise the run-time code [17]. Dynamic software updating modifies possibly stateful software to fix bugs and add functionality without shutting that software down, and guarantees some safety property to hold over the update boundary [24,38,38,45]. Program generation is a similar problem, and is commonly used in dynamic scripting languages such as Perl and Python. In order to extend type and effect systems to languages which permit dynamic modifications we must permit the effect of code to likewise be modified.

When making a dynamic modification the programmer will want preserve previous behavioural properties after the modification. We refer to an update which maintains the required behavioural properties as a *behaviour safe* update. An example is installing drivers or extensions into an operating system kernel where any instabilities in the extension should not affect the rest of the kernel. This is often enforced by requiring the extension, whilst being in the same address space as the kernel, to not access any memory except its own. This behavioural property can be represented in an effect system tracking memory accesses [45], or using access policies.

We first approach behaviour safe updates for variable type access systems (Section 5.2). We present our plans for continuing our research on MDSU (Section 5.3). We conclude in Section 5.4

5.2 Dynamic Modifications for Variable Type Access Systems

In this section we describe how, in order to reason about the behavioural safety of an update, we need to know the effect of the runtime code and the current state of the resources [1]. This is as variable type safe access of resource depends on the state of the resources as well as the effect of the code accessing those resources (Example 3.2). We present infrastructure for modifying this effect and reasoning about its modified form (Section 5.2.1). We present two possible approaches for obtaining the effect of the runtime code besides doing a full type and effect analysis on the runtime code (Section 5.2.3). We show how this work can be leveraged to give a runtime check of behavioural update safety (Section 5.2.2).

5.2.1 Modifying Effects

In order to reason about the behavioural safety of dynamic modifications we need to be able to reason about how some code's effect can change at runtime. This depends on the mechanism used for modifying the code.

5.2.1.1 Effect Annotation

The most common approach is to keep function definitions on the heap and to substitute them into the code at their call sites rather than through the entire program after definition (which is the standard functional programming approach) [24, 25, 37, 39, 45]. As the function definitions are on the heap the update mechanism simply has to change these definitions, and from that point onwards the new definition will be substituted whenever the function is called. This approach does not change the code of a function which is being evaluated when the update occurs. For example:

Example 5.1.

$$\begin{aligned}
&(\mathsf{f}(x_1, x_2) \mapsto e_f, \mathsf{g}() \mapsto e_g), \mathsf{f}(2, \boldsymbol{\mathit{true}}); \boldsymbol{\mathit{update}}; \mathsf{g}() \\
&\quad \rightarrow (f \mapsto e_f, g \mapsto e_g), e_f[2/x_1][\boldsymbol{\mathit{true}}/x_2]; \boldsymbol{\mathit{update}}; \mathsf{g}() \\
&\quad \textit{introduce an update which maps } f \textit{ to } e'_f \textit{ and } g \textit{ to } e'_g \\
&\quad \rightarrow (\mathsf{f}(x_1, x_2) \mapsto e'_f, \mathsf{g}() \mapsto e'_g), e_f[2/x_1][\boldsymbol{\mathit{true}}/x_2]; \boldsymbol{\mathit{update}}; \mathsf{g}() \\
&\quad \ldots \\
&\quad \rightarrow (\mathsf{f}(x_1, x_2) \mapsto e'_f, \mathsf{g}() \mapsto e'_g), \mathsf{g}() \\
&\quad \rightarrow (\mathsf{f}(x_1, x_2) \mapsto e'_f, \mathsf{g}() \mapsto e'_g), e'_g
\end{aligned}$$

Note that the inlined body of f is not changed by the update.

A generalisation of this approach is to include named regions of code. These annotations in the source code denote regions of code which can be replaced in their entirety e.g.:

$$3 + 4;\ main(\texttt{acc}^{l_1};\ \texttt{acc}^{l_2});\ \texttt{acc}^{l_3} \tag{5.1}$$

where $main$ is a name and $\texttt{acc}^{l_1};\ \texttt{acc}^{l_2}$ is the annotated region. Updates then consist of mappings from annotation names to new region bodies, e.g. $(main \mapsto \texttt{acc}^{l_4};\ \texttt{acc}^{l_2})$, and the update mechanism consists of doing the substitution of $main(e')$ for $main(\_)$.

This approach alone, however, is not suitable for modifying code with side effects, as we have no connection between some code and its effect without doing a type and effect analysis. For example, the code in Equation 5.1 has the effect:

$$L_1;\ L_2;\ L_3 \tag{5.2}$$

where L_i represents l_i. We cannot tell, by looking at the effect alone, the effect which corresponds with the annotated code. A similar problem would occur if we used the function-specific DSU approach above. This is a problem if we want to see how an update modifies the effect of some code without having to redo the entire effect analysis.

Our solution to this problem is to include annotations on the effect as well as the code. This links the code of an annotated regions to its effect. Assuming we have the effect of some running code, suitably annotated, we can use this infrastructure to determine what the effect of some updated code is without having to redo the effect analysis for the modified code (Section 5.2.1.3).

5.2.1.2 Position of Updates

It is possible to apply an update to every annotated region as soon as it becomes available. This has two problems: reasoning about updates, and proving their safety.

As discussed in Section 2.4 with respects to Figure 2.1 if an update occurs in an inopportune place, generally anywhere besides the start of the main loop or a dedicated function for performing updates, the programmer's reasoning about what it will do can be broken. In the case of the example in Figure 2.1 a log entry can be lost. We make use of a language construct $\texttt{dmod}(e)$ which denotes that updates can occur to code within the construct. Intuitively this means that for the code:

$$e_1;\ \texttt{dmod}(e_2)$$

that an update will not be visible in the execution of this program until it reaches the code e_2. This approach is a slight modification of the standard DSU technique of using an `update` keyword to denote where an update will occur [13, 16, 25, 39, 45]. We do this in order to aid syntactic reasoning about where and when updates to occur within code's effect.

It is possible to use transactional techniques to obtain a useful reasoning paradigm for updates and more liberality on where updates can be applied to code, for simple type updates [38]. This may be useful for extending our approach to a static analysis, and is part of our proposed future work (Section 5.3).

5.2.1.3 Formal Definitions

We define a simple lambda-calculus with recursive functions and a single effectful primitive $\texttt{acc}^l$ such as in [32, 40]. We also include the language construct $\texttt{dmod}(e)$ which we use to syntactically define where and when a system may perform dynamically modifications, and $name(e)$ which denotes a section of code which can be replaced in its entirety. Hence expressions and effects are defined as:

Definition 5.1.

$$
\begin{array}{rll@{\qquad}rll@{\qquad}rll}
e ::= & v & & v ::= & n & & \varphi ::= & \epsilon \\
| & \texttt{rec}\, f{=}\lambda x.e & & | & b & & | & (L,T) \\
| & e\,e & & | & () & & | & \varphi;\varphi \\
| & \texttt{acc}^l & & | & r & & | & \varphi{+}\varphi \\
| & \texttt{if}\; e \;\texttt{then}\; e \;\texttt{else}\; e & & | & x & & | & \varphi\&\varphi \\
| & \texttt{acc}^l\&(T_1{\Rightarrow}e_1,\ldots,T_n{\Rightarrow}e_n) & & & & & | & \mu\mathbf{t}.\varphi \\
| & \texttt{dmod}(e) & & P ::= & \langle e\rangle & & | & \texttt{dmod}(\varphi) \\
| & name(e) & & | & P \| P & & | & name(\varphi)
\end{array}
$$

We refer to parallel code as *well formed* if none of the annotation names used by one thread are used by any other. We assume code is well formed. The current state of the shared resource(s) and the multi-threaded code which has yet to be executed is referred to as a *snapshot* of a system. We add additional reduction and type rules for the new constructs (Figure 5.2).

An update is a set of mappings from annotation names to expressions:

Definition 5.2.

$$
\begin{array}{rl}
\mu ::= & name{\mapsto}e \\
| & \mu,\mu
\end{array}
$$

where updates are well formed with respects to some code if the simple types of the replacement and the old code are the same. We assume well formed updates. We define the update mechanism in Figure 5.1. The boolean in this function denotes whether the code or effect in the argument is syntactically within a $\texttt{dmod}$ construct. Intuitively these

$$\mathsf{upd}(P,\mu,b) \overset{\text{def}}{=} \begin{cases} \mathsf{upd}(P_1,\mu,\texttt{false}) \parallel \mathsf{upd}(P_2,\mu,\texttt{false}) & P = P_1 \parallel P_2 \\ P & \Phi = (L,T) \vee \epsilon \\ \mathsf{upd}(e_1,\mu,b);\ \mathsf{upd}(e_2,\mu,b) & P = e_1;\ e_2 \\ \mathsf{upd}(e_1,\mu,b) + \mathsf{upd}(e_2,\mu,b) & P = e_1 + e_2 \\ \mathsf{upd}(e_1,\mu,b)\&\mathsf{upd}(e_2,\mu,b) & P = e_1\&e_2 \\ \mu\mathbf{t}.\mathsf{upd}(e,\mu,b) & P = \mu\mathbf{t}.e \\ \texttt{dmod}(\mathsf{upd}(e,\mu,\texttt{true})) & P = \texttt{dmod}(e) \\ name(\mathsf{upd}(e,\mu,b)) & P = name(e) \wedge \\ & (b = \texttt{false} \vee name \notin \mathsf{dom}(\mu)) \\ name(\mu(name)) & P = name(e) \wedge b = \texttt{true} \wedge \\ & name \in \mathsf{dom}(\mu) \end{cases}$$

$$\mathsf{upd}(\Phi,\mu,b) \overset{\text{def}}{=} \begin{cases} \mathsf{upd}(\Phi_1,\mu,\texttt{false}) \parallel \mathsf{upd}(\Phi_2,\mu,\texttt{false}) & \Phi = \Phi_1 \parallel \Phi_2 \\ \Phi & \Phi = (L,T) \vee \epsilon \\ \mathsf{upd}(\varphi_1,\mu,b);\ \mathsf{upd}(\varphi_2,\mu,b) & \Phi = \varphi_1;\ \varphi_2 \\ \mathsf{upd}(\varphi_1,\mu,b)\&\mathsf{upd}(\varphi_2,\mu,b) & \Phi = \varphi_1 + \varphi_2 \\ \mathsf{upd}(\varphi_1,\mu,b)\&(\mathsf{upd}(\varphi_2,\mu,b)) & \Phi = \varphi_1\&\varphi_2 \\ \mu\mathbf{t}.\mathsf{upd}(\varphi,\mu,b) & \Phi = \mu\mathbf{t}.\varphi \\ \texttt{dmod}(\mathsf{upd}(\varphi,\mu,\texttt{true})) & \Phi = \texttt{dmod}(\varphi) \\ name(\mathsf{upd}(\varphi,\mu,b)) & \Phi = name(\varphi) \wedge \\ & (b = \texttt{false} \vee name \notin \mathsf{dom}(\mu)) \\ name(\varphi') & \Phi = name(\_) \wedge b = \texttt{true} \wedge \\ & name \in \mathsf{dom}(\mu) \wedge \\ & \emptyset \vdash \mu(name) : T \rhd \varphi' \end{cases}$$

Figure 5.1: Updating Code and Effects

$$\frac{}{[\sigma]\ name(e) \longrightarrow [\sigma]\ e}\ (\text{RAnnot}) \qquad \frac{}{[\sigma]\ \texttt{dmod}(e) \longrightarrow [\sigma]\ e}\ (\text{RDmod})$$

$$\frac{}{[\sigma]\ P \longrightarrow [\sigma]\ \mathsf{upd}(P,\mu,\texttt{false})}\ (\text{RUpdNaive})$$

$$\frac{\varphi \wr \Gamma \vdash e \colon T}{name(\varphi) \wr \Gamma \vdash name(e) \colon T}\ (\text{:TAnnot}) \qquad \frac{\varphi \wr \Gamma \vdash e \colon T \quad \mathsf{fv}(e) = \emptyset}{\texttt{dmod}(\varphi) \wr \Gamma \vdash \texttt{dmod}(e) \colon T}\ (\text{:TDmod})$$

Figure 5.2: Additional Rules for Updateable Systems

functions replace annotated code sections if they are within a `dmod` construct and there is an update for that annotation. For example:

Example 5.2.

$$\begin{aligned} &\mathsf{upd}(f(e_1);\ \texttt{dmod}(f(e_1);\ g(e_2)), (f \mapsto e_3), \texttt{false}) \\ &\quad = \mathsf{upd}(f(e_1),\mu,\texttt{false});\ \mathsf{upd}(\texttt{dmod}(f(e_1);\ g(e_2)),\mu,\texttt{false}) \\ &\qquad\quad \textit{where } \mu = (f \mapsto e_3) \\ &\quad = f(\mathsf{upd}(e_1,\mu,\texttt{false}));\ \mathsf{upd}(\texttt{dmod}(f(e_1);\ g(e_2)),\mu,\texttt{false}) \\ &\quad = f(\mathsf{upd}(e_1,\mu,\texttt{false}));\ \texttt{dmod}(\mathsf{upd}(f(e_1);\ g(e_2),\mu,\texttt{true})) \\ &\quad = f(e_1);\ \texttt{dmod}(f(e_3);\ g(e_2)) \end{aligned}$$

where there are no annotated regions in e_1,e_2. Note that this replaces the annotation associated with f inside the `dmod`, but not the one outside it. This corresponds to our intuition that an update does not become visible until we encounter a `dmod` construct. We can then prove that this transformation maintains the link between code and its effect.

Conjecture 5.3. **Effect Safe Update**

If $\Phi \wr \Gamma \vdash P\colon T$ and μ is well formed with respects to P then

$$\mathsf{upd}(\Phi, \mu, \mathtt{false}) \wr \Gamma \vdash \mathsf{upd}(P, \mu, \mathtt{false})\colon T$$

5.2.2 Runtime Safety Check

In order to guarantee that all accesses are type safe we do model checking, or some check which implies model checking, of the property that irrespective of interleaving of accesses that the same type is always returned. When we modify code we also modify its effect. Hence in order to prove that the modified code is still access safe we need to redo model checking on the new effect. Specifically, in order to guarantee that the update rule:

$$\frac{}{[\sigma]\, P \longrightarrow [\sigma]\, \mathsf{upd}(P,\mu,\mathtt{false})}\ (\textsc{RUpdNaive})$$

respects Subject Reduction (Theorem 3.3.4.1):

$$\vdash P\colon \Phi \wedge [\sigma]\, P \rightarrow [\sigma']\, P' \Rightarrow \vdash P'\colon \Phi' \wedge \Phi \xrightarrow{-} \Phi'$$

we have to show that:

$$\vdash P\colon \Phi \wedge \sigma\colon \Sigma \wedge \mathsf{compatible}(\mathsf{upd}(\Phi, \mu, \mathtt{false}), \Sigma)$$

Intuitively this requires that the updated effect be compatible with the current state of the resources. We hence extend the update reduction rule to include this runtime safety check:

$$\frac{\sigma\colon \Sigma \quad \vdash P\colon \Phi \quad \mathsf{compatible}(\mathsf{upd}(\Phi,\mu,\mathtt{false}),\Sigma)}{[\sigma]\, P \longrightarrow [\sigma]\, \mathsf{upd}(P,\mu,\mathtt{false})}\ (\textsc{RUpd})$$

As, by Theorem 5.3, $\mathsf{upd}(\Phi, \mu, \mathtt{false})$ correctly represents the effect of $\mathsf{upd}(P, \mu, \mathtt{false})$, if we can show that the modified effect respects compatibility then the modified code will respect subject reduction.

5.2.3 Obtaining the Effect of Runtime Code

The runtime check in Section 5.2.2 requires that we know the effect of the code being update. We do not want to have to redo the effect analysis at runtime as this would

$$\frac{[\sigma],e_1,\varphi_1 \longrightarrow [\sigma'],e_1',\varphi_1'}{[\sigma],e_1\ e_2,\varphi_1;\varphi_2 \longrightarrow [\sigma'],e_1'\ e_2,\varphi_1';\varphi_2}\ (\textsc{RMAppOne}) \qquad \frac{[\sigma],e_2,\varphi_2 \longrightarrow [\sigma'],e_2',\varphi_1'}{[\sigma],v\ e_2,\varphi_2 \longrightarrow [\sigma'],v\ e_2',\varphi_2'}\ (\textsc{RMAppTwo})$$

$$\frac{}{[\sigma],\mathtt{rec} f{=}\lambda x.e\ v,\varphi \longrightarrow [\sigma],e[\mathtt{rec} f{=}\lambda x.e/f][v/x],\varphi}\ (\textsc{RMAppThree})$$

$$\frac{\sigma \xrightarrow{l} \sigma'}{[\sigma],\mathtt{acc}^l,(L,T) \longrightarrow [\sigma'],\sigma(l),\epsilon}\ (\textsc{RMAcc}) \qquad \frac{}{[\sigma],name(e),name(\varphi) \longrightarrow [\sigma],e,\varphi}\ (\textsc{RMAnnot})$$

$$\frac{[\sigma],e_1,\varphi_1 \longrightarrow [\sigma'],e_1',\varphi_1'}{[\sigma],\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3,\varphi_1;\varphi_2{+}\varphi_3 \longrightarrow [\sigma'],\mathtt{if}\ e_1'\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3,\varphi_1';\varphi_2{+}\varphi_3}\ (\textsc{RMIfOne})$$

$$\frac{}{[\sigma],\mathtt{if\ true\ then}\ e_2\ \mathtt{else}\ e_3,\varphi_2{+}\varphi_3 \longrightarrow [\sigma],e_2,\varphi_2}\ (\textsc{RMIfTwo})$$

$$\frac{}{[\sigma],\mathtt{if\ false\ then}\ e_2\ \mathtt{else}\ e_3,\varphi_2{+}\varphi_3 \longrightarrow [\sigma],e_3,\varphi_3}\ (\textsc{RMIfThree})$$

$$\frac{\sigma \xrightarrow{l} \sigma' \qquad \sigma(l) : T_i}{[\sigma],\mathtt{acc}^l\&(T_1{\Rightarrow}e_1,\ldots,T_n{\Rightarrow}e_n),(L,T_1);\varphi_1\&\ldots\&(L,T_n);\varphi_n \longrightarrow [\sigma'],e_i,\varphi_i}\ (\textsc{RMExtChoice})$$

$$\frac{[\sigma],e,\varphi \longrightarrow [\sigma'],e',\varphi'}{[\sigma],\langle e\rangle,\varphi \longrightarrow [\sigma'],\langle e'\rangle,\varphi'}\ (\textsc{RMProc})$$

$$\frac{[\sigma],P_1,\Phi_1 \longrightarrow [\sigma'],P_1',\Phi_1'}{[\sigma],P_1 \| P_2,\Phi_1 \| \Phi_2 \longrightarrow [\sigma'],P_1' \| P_2,\Phi_1' \| \Phi_2}\ (\textsc{RMParOne})$$

$$\frac{[\sigma],P_2,\Phi_2 \longrightarrow [\sigma'],P_2',\Phi_2'}{[\sigma],P_1 \| P_2,\Phi_1 \| \Phi_2 \longrightarrow [\sigma'],P_1 \| P_2',\Phi_1 \| \Phi_2'}\ (\textsc{RMParTwo})$$

$$\frac{\sigma : \Sigma \qquad \mathsf{compatible}(\mathsf{upd}(\Phi,\mu,\mathtt{false}),\Sigma)}{[\sigma],P,\Phi \longrightarrow [\sigma],\mathsf{upd}(P,\mu,\mathtt{false}),\mathsf{upd}(\Phi,\mu,\mathtt{false})}\ (\textsc{RMUpd})$$

Figure 5.3: Semantics For Runtime Modelling

mean suspending the system for longer than we would like. Instead we would like to be able to obtain the effect of the current code by some other method. We refer to methods for obtaining the effect of runtime code as *effect capture methods.*

Fully model checking the compatibility property would also requiring suspending the system for infeasible lengths of time. If, however, we are using some linear-sized predicate check on the effect instead of model checking then the suspension time may be acceptable.

5.2.3.1 Runtime Modelling

We can take the effect from the effect analysis when the code is first analysed (before the update) and evolve it along with the code at runtime. We present a semantics which reduces the effect with the code in Figure 5.3. We can then prove subject reduction as in Section 5.2.2. The advantages of this approach are that the effect is instantly available and has minimal space requirements. The disadvantage is the computation cost of running the model.

$$\frac{x\colon T \in \Gamma}{\emptyset \wr \varphi \wr \Gamma \vdash x\colon T}\text{(:TAVar)} \qquad \frac{l\colon L \quad \varphi' = (L,T);\ \varphi}{(L,T) \wr \varphi' \wr \Gamma \vdash \mathtt{acc}^l\{\varphi'\}\colon T}\text{(:TAAcc)}$$

$$\frac{\varphi \wr \Gamma, x\colon T_1, f\colon T_1 \xrightarrow{\varphi} T_2 \vdash e\colon T_2}{\emptyset \wr \Gamma \vdash \mathtt{rec} f = \lambda x.e\colon T_1 \xrightarrow{\varphi} T_2}\text{(:TALam)} \qquad \frac{\varphi_1 \wr \varphi_2;\varphi_3;\varphi \wr \Gamma \vdash e_1\colon T_1 \xrightarrow{\varphi_3} T_2 \quad \varphi_2 \wr \varphi_3;\varphi \wr \Gamma \vdash e_2\colon T_1}{\varphi_1;\ \varphi_2;\ \varphi_3 \wr \varphi \wr \Gamma \vdash e_1\, e_2\colon T_2}\text{(:TAApp)}$$

$$\frac{\varphi_1 \wr \varphi_2 \oplus \varphi_3; \varphi \wr \Gamma \vdash e_1\colon \textsc{Bool} \quad \varphi_2 \wr \varphi \wr \Gamma \vdash e_2\colon T \quad \varphi_3 \wr \varphi \wr \Gamma \vdash e_3\colon T}{\varphi_1;\ (\varphi_2 \oplus \varphi_3) \wr \varphi \wr \Gamma \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\colon T}\text{(:TAIfExpr)}$$

$$\frac{\varphi_i \wr \varphi' \wr \Gamma \vdash e_i\colon T \quad \Gamma \vdash l\colon L \quad \varphi = (L,T_1);\ \varphi_1 \& ... \& (L,T_n);\ \varphi_n}{\varphi \wr \varphi' \wr \Gamma \vdash \mathtt{acc}^l \& (T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n)\{\varphi;\ \varphi'\}\colon T}\text{(:TAExtChoice)}$$

$$\frac{l = \alpha(v_1, ..., v_n) \qquad L = \alpha(T_1, ..., T_n) \qquad \emptyset \wr \Gamma \vdash v_i\colon T_i}{\Gamma \vdash l\colon L}\text{(:TALab)}$$

$$\frac{\varphi \wr \varphi' \wr \Gamma \vdash e\colon T \quad \mathsf{fv}(e) = \emptyset}{\mathit{name}(\varphi) \wr \varphi' \wr \Gamma \vdash \mathit{name}(e)\colon T}\text{(:TAAnnot)}$$

$$\frac{\varphi \wr \varphi' \wr \Gamma \vdash e\colon T \quad \mathsf{fv}(e) = \emptyset}{\mathtt{dmod}(\varphi) \wr \varphi' \wr \Gamma \vdash \mathtt{dmod}(e)\{\mathtt{dmod}(\varphi); \varphi'\}\colon T}\text{(:TAModDef)}$$

$$\frac{\varphi_i \wr \emptyset \wr \emptyset \vdash e_i\colon T_i \quad \sigma\colon \Sigma \quad \mathsf{compatible}(\varphi_1 \parallel ... \parallel \varphi_n, \Sigma)}{\vdash \langle e_1 \rangle \parallel ... \parallel \langle e_n \rangle\colon \varphi_1 \parallel ... \parallel \varphi_n}\text{(:TAPar)}$$

$$\frac{\vdash P\colon \Phi \quad \sigma\colon \Sigma \quad \mathsf{compatible}(\Phi, \Sigma)}{\sigma \vdash P\colon \Phi}\text{(:ValidThreads)} \quad \text{where} \quad \begin{array}{l}\mathsf{compatible}(\Phi,\Sigma)\\ \Rightarrow \mathsf{compGen}(\Phi,\Sigma)\end{array}$$

FIGURE 5.4: Annotating Type Rules

5.2.3.2 Annotation

It is possible to, during the effect analysis, transform the code into a runtime format which annotates the effect of the code from that point onwards onto constructs relevant to updates.

We redefine the access, external choice, and update point constructs:

Definition 5.4.

$$\begin{array}{rl} e ::= & ... \\ | & \mathtt{acc}^l \& (T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n)\{\varphi\} \\ | & \mathtt{acc}^l\{\varphi\} \\ | & \mathtt{dmod}(e)\{\varphi\} \end{array}$$

The effect annotated on these constructs denotes the effect of this construct and of all code following it. This technique uses the effect system to 'look ahead', and this 'future effect' is annotated on the construct. This is a use of the contextual effects approach [38]. We redefine the effect system in Figure 5.4. We extend our normal type and effect judgement to:

$$\varphi \wr \varphi' \wr \Gamma \vdash e\colon T$$

which denotes that, under type variable assumptions Γ, an expression e has simple type T and its evaluation is approximated by φ. Additionally, φ' denotes the effect of code following the expression. Note that, in the majority of rules, this effect is not defined, it is simply stated; we cannot determine what follows after the construct as we don't have that code. Instead we determine this effect by unification, using the two rules which place constraints on how the future effect is constructed. The TAApp rule defines that the effect of the code following e_1 consists of the effect of e_2, the latent effect of the function which e_1 evaluates to, followed by the effect of the code following the entire expression. The effect of the code following e_2 consists of the latent effect of the function e_1 evaluates to and the effect the code following the entire expression. The TAPar rule stipulates that there is no effect of code following the body of a thread ($\varphi_i \wr \mathbf{\emptyset} \wr \emptyset \vdash e_i \colon T_i$) as there is no code following the body of a thread.

When we use annotation all we need to do to determine the effect of some code is search for the next annotated statement and extract the effect:

Definition 5.5.

$$\mathsf{extract}(P) \overset{\text{def}}{=} \begin{cases} \epsilon & P = v \vee \mathtt{rec} f = \lambda x.e \\ \mathsf{extract}(e_1) & P = e_1\, e_2 \wedge \mathsf{extract}(e_1) \neq \epsilon \\ \mathsf{extract}(e_2) & P = e_1\, e_2 \wedge \mathsf{extract}(e_1) = \epsilon \\ \mathsf{extract}(e_1) & P = \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \wedge \mathsf{extract}(e_1) \neq \epsilon \\ \mathsf{extract}(e_2) + \mathsf{extract}(e_3) & P = \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \wedge \mathsf{extract}(e_1) = \epsilon \\ \varphi & P = \mathtt{acc}^l\{\varphi\} \\ \varphi & P = \mathtt{acc}^l\&(T_1 \Rightarrow e_1, ..., T_n \Rightarrow e_n)\{\varphi\} \\ \varphi & P = \mathtt{dmod}(e)\{\varphi\} \\ \mathsf{extract}(e) & P = name(e) \\ \mathsf{extract}(P_1) \| \mathsf{extract}(P_2) & P = P_1 \parallel P_2 \end{cases}$$

We could annotate all constructs with the future effects. The effect is only defined, however, by the access, external choice, region annotation, and dynamic modification points; between the other constructs the effect is not changed and it would expand the code unnecessarily to annotate these points. We do not annotate the 'region annotation' construct as:

$$\mathsf{upd}(name(e), \mu, \mathtt{false}) = \mathsf{upd}(e, \mu, \mathtt{false})$$

Unless we are within a dynamic modification point when using the update function, we ignore the region annotation. If an annotation region is outside a dynamic modification including the region annotation makes no difference and we can continue inside the region to the next relevant construct. If it is within a dynamic modification construct then we would have already hit that when extracting the effect. Hence we can safely not annotate named regions with the future effect.

We posit that this extracted effect comprises the code's effect (the proof is future work):

Conjecture 5.6. **Correct Effect Annotation**

If P is code which has been annotated by a prior effect analysis then $\vdash P\colon \mathsf{extract}(P)$

Assuming Lemma 5.6 is correct then we can define a semantics which is the same as in Section 5.2.1.3 except for with a modified update reduction:

$$\frac{\sigma : \Sigma \qquad \mathsf{compatible}(\mathsf{upd}(\mathsf{extract}(P),\mu,\mathtt{false}),\Sigma)}{[\sigma]\, P \longrightarrow [\sigma]\, \mathsf{upd}(P,\mu,\mathtt{false})}\ (\textsc{RAUpd})$$

We can then prove subject reduction as in Section 5.2.2. The advantage of this approach is that it doesn't require running a model of the effect as well as the code. The disadvantage is that code size is increased as we annotate the future effect everywhere it could be relevant.

5.3 Future Work

We present a series of possible problems related to DSU which we hope to cover during the remainder of the thesis. In particular these cover approaches which will allow us to develop a static rather than dynamic analysis, technique of covering different types of updates, and efficiency issues.

5.3.1 Dynamic Modifications for Access Policy Systems

Both the resource access with variably typed return and the access policy systems use model checking on the effect of code. Hence we posit that it would be possible, in theory to use the same approach for access policies. Namely this approach is to annotate the code with updateable regions, use an effect capture method to obtain the runtime effect, modify it using the update mechanism, and then use the model checking techniques for multi-threaded programs using access policies (Chapter 4). Unfortunately, as model checking tends to be a time-consuming process, this is probably infeasible for a runtime check. Hence in order to update access policy systems we need to do a static analysis.

5.3.2 Comparing Effect Capture Methods

The runtime modelling method evaluates the effect of the code, determined by an initial effect analysis, alongside the evaluation of the code. This has the obvious advantage that the effect is instantly available (we need not delve through the code to find it on an annotation), and compared to the annotation model it requires little space. Effectively it requires one copy of the code's effect to be recorded (and updated). The annotation method effectively requires us to record the entire effect of the code every time the effect changes; intuitively this seems be exponential in the size of the effect. The disadvantage of this approach is running the effect alongside the code, specifically ensuring they evaluate in lockstep. In practice this would probably require mutual exclusion, which would slow down the system.

The main advantage of the annotation method is that it does not require evaluating the effect in lockstep, which would eliminate the slow-down related to mutual exclusion. It might, however, introduce a slow-down related to the annotations; the runtime processor would need to deal with the additional, normally unused, annotations of code. Additionally, the number of annotations may make the code infeasibly large. If, however, we can use transactional techniques to reduce the places where effects can be visibly introduced (Section 5.3.3) we may be able to reduce the number of places we need to annotate the code.

We propose to, after developing an example implementation of our system, run benchmarks to compare these two methods. We also propose to investigate using transactional techniques to reduce the number of annotation points.

5.3.3 Transactional Update Approach and Static Analyses

In [37] the authors use an approach which determines induced update points which are program points p such that if an update is applied when a thread reaches p, the program will behave as if the patch had been applied at some other update point p'. The authors also use transactional techniques to denote regions which must appear to have run with entirely the old code or entirely the new code. This is similar to where transactions appear to have occurred using exactly one version of a database. The induced update points allow the authors to determine entire regions where updates can be applied and appear as if the update occurred at a specific, later point. These 'safe' regions are determined threadwise. The innovative technique for MDSU in [37] is simply to apply the update when all threads are in safe regions, and not to try to force them to synchronise on these regions in any way. This reduces the delay cost of requiring threads to synchronise for an update and the possibility of deadlocking whilst waiting. Additionally these points can be statically determined for each thread, so it is statically possible to know whether there could, theoretically, be any overlap in points where all can safely perform the updates (i.e. whether the threads' respective safe regions are all safe for the aspects of the code being updated). We hope to be able to extend our work to similarly use transactional techniques which would allow a fully static analysis.

5.3.4 Updates Dependent on State of Accompanying Resources

In some situations we may wish to apply an update for the entire system to different threads at different times. We explain with an example; consider a producer-consumer system:

$$\varphi_p || \varphi_c$$

$$\varphi_p = \mu\underline{\mathsf{h}}.\mathtt{dmod}(f((snd(\mathsf{Res}\,c, \textsc{Int}), \textsc{Unit}); (snd(\mathsf{Res}\,c, \textsc{Int}), \textsc{Bool}); (snd(\mathsf{Res}\,c, \textsc{Int}), \textsc{Unit}); \underline{\mathsf{h}}))$$
$$\varphi_c = \mu\underline{\mathsf{h}}'.\mathtt{dmod}(g((rcv(\mathsf{Res}\,c), \textsc{Int}); (rcv(\mathsf{Res}\,c), \textsc{Bool}); (rcv(\mathsf{Res}\,c), \textsc{Int}); \underline{\mathsf{h}}'))$$

In this example we have two threads, one which repeatedly sends an integer, followed by a boolean, followed by another integer, and one which receives values of these types. Using standard asynchronous message passing semantics the producer is never blocked by the receiver, and can 'speed ahead' and send more triples of data than the receiver has received. Hence we can end up with a situation where the state system resembles:

$$[c \mapsto \textsc{Int}, \textsc{Int}, \textsc{Bool}, \textsc{Int}, \textsc{Int}, \textsc{Bool}]\ \varphi_p || \varphi_c$$

i.e. there are two iterations worth of data waiting for the consumer. Consider an update which changes the system so that there are only two values sent per iteration:

$$\begin{aligned} \mu = \quad & f \mapsto (snd(\mathsf{Res}\,c, \textsc{Int}), \textsc{Unit}); (snd(\mathsf{Res}\,c, \textsc{Int}), \textsc{Bool}), \\ & g \mapsto (rcv(\mathsf{Res}\,c), \textsc{Int}); (rcv(\mathsf{Res}\,c), \textsc{Bool}) \end{aligned}$$

If we attempt to apply this update to the system we will cause an error as the consumer will attempt to do the actions:

$$(rcv(\mathsf{Res}\,c), \textsc{Int}); (rcv(\mathsf{Res}\,c), \textsc{Bool})$$

on a resource:

$$c \mapsto \textsc{Int}, \textsc{Int}, \ldots$$

This update is not, however, fundamentally incompatible with this system. We could apply the update to the producer immediately, and only apply it to the consumer once it has consumed all the data sent to it under the old protocol. We can represent this by including with each region update a predicate p on the state of the shared resources, i.e.

$$\begin{aligned} \mu ::= \quad & name{\mapsto}(e{,}p) \\ | \quad & \mu{,}\mu \end{aligned}$$

We would then need to modify the update mechanism to, rather than applying the update immediately when it is obtained, wait until `dmod` constructs are encountered, to check whether the predicate is fulfilled, and if so to apply the update to the code within the `dmod`. This would require retaining update definitions in conjunction with the accompanying shared resources in the runtime configuration: $[\sigma], P, \mu$. In such a system we would remove the RUpd rule and modify the RDmod rule into:

$$\frac{\begin{array}{c}\mu = \mu', \mu'' \\ \forall name \in \mathsf{dom}(\mu').\mu(name) = (\_, p) \wedge p(\sigma) = \mathtt{true} \\ \forall name \in \mathsf{dom}(\mu'').\mu(name) = (\_, p) \wedge p(\sigma) = \mathtt{false}\end{array}}{[\sigma], \mathtt{dmod}(e), \mu \longrightarrow [\sigma], \mathsf{upd}(e, \mu', \mathtt{true}), \mu''} \text{(RDDmod)}$$

Conceptually this removes the boolean check as to whether the update function is within a `dmod` as updates now always occur at RDmod constructs, and adds the semantics that we only perform updates which have a valid predicate in the current state.

We should still be able to, when an update is introduced, model check to see whether applying the update will lead to violating subject reduction or not. We could also use similar techniques of simpler predicates and invariability proofs as well as transaction approaches for static analyses.

5.4 Conclusions

We develop infrastructure to permit us to determine the effect of some updated code using the runtime effect of some runtime code by using code and effect annotations. We show what properties we need in order to guarantee that an update preserves Subject Reduction. We explore possible effect capture methods to determine the effect of runtime code. We present future work which includes approaches to permit more permissive updates whose application are dependent on the runtime state of the resources, and techniques which may permit us to create a static analysis with minimum runtime costs.

Chapter 6

Conclusions

6.1 Plan

We present our future work by topic below. We present our timetable for the remainder of the Ph.D. in Figure 6.1.

6.1.0.1 Resource Access with Variably Typed Return

We plan to explore invariability predicates and invariability proofs (Section 3.4). We expect that there are common aspects of these predicates and proofs which we may be able to extract and prove more generally, in order to reduce even more the burden of proof on the resource system architect. We intend to incorporate the approach taken to dynamic resources and model checking properties using them from [7] into variably typed return systems.

Month	Activity
25	Formalise DSU for access policy systems
26	Explore examples of and themes occurring in invariability proofs
27	Incorporate dynamic resources into variably typed return systems
28-29	Transactional approach to updates and static analysis
30	Integrating model checking techniques from variably typed return and access policy systems
31-32	Implementation
33-36	Write up Thesis
37	Submit Thesis

FIGURE 6.1: Work plan for the remainder of the Ph.D.

6.1.0.2 Dynamic Software Updating

We will to formalise our intuition about DSU for access policy systems. We intend to expand our approach to incorporate updates which are dependent on the state of the shared resources. We hope to use transactional techniques demonstrated in [37] to create an update mechanism which requires less synchronisation but also has fewer possible interleavings with respects to when updates can occur and hence make a static analysis feasible. After doing our implementation we will benchmark and compare our different effect capture systems.

6.1.0.3 Other

We intend to provide a sample implementation of our functional concurrent language which makes use of shared resources. We will implement the analyses designed in this thesis which guarantee the type safe use of the shared resources and that the code's effect respects the access policies. We will implement the dynamic safety checks and hopefully a static analysis to guarantee that updates do not break the proved behavioural properties.

6.2 Closing Thoughts

We use the established technique of model checking to prove different formal behavioural properties for concurrent programs. We generalise session typing approaches to reveal the essence of their approaches and to simplify their proofs. We expand access policy techniques to cover multi-threaded programs. We show how to modify running concurrent programs whilst maintaining behavioural safety properties using dynamic safety checks. We posit how to extend our approach to encompass more expressive updates and how to develop a fully static analysis.

Bibliography

[1] Austin Anderson and Julian Rathke. Migrating protocols in multi-threaded message-passing systems. *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*, page 1, 2009.

[2] J.L. Armstrong and S.R. Virding. ERLANG - an experimental telephony programming language.

[3] Massimo Bartoletti. Usage Automata. pages 52–69, 2009.

[4] Massimo Bartoletti, P Degano, G Ferrari, and Roberto Zunino. 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, 2007.

[5] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. *Types and Effects for Resource Usage Analysis*, pages 32–47. Springer Berlin / Heidelberg, 4423/2007 edition, 2007.

[6] Massimo Bartoletti, Pierpaolo Degano, Gian-Luigi Ferrari, and Roberto Zunino. Local policies for resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 31(6):1–43, August 2009.

[7] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. *Model Checking Usage Policies*, volume Trustworth, chapter Model Chec, pages 19–35. Springer-Verlag, 2009.

[8] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. *ACM SIGPLAN Notices*, 40(6):305, June 2005.

[9] Savage Becker, Brian Marc, E N Bershad Fiuczynski, Stefan David, and Science Washington. Extensibility safety and performance in the SPIN operating system. *System*, pages 267–284, 1995.

[10] Andi Bejleri and Nobuko Yoshida. Synchronous Multiparty Session Types. *Electron. Notes Theor. Comput. Sci.*, 241:3–33, July 2009.

[11] Lorenzo Bettini, Mario Coppo, Marco De Luca, Mariangiola Dezani-ciancaglini, and Nobuko Yoshida. *Global Progress in Dynamically Merged Multiparty Sessions*, pages 418–433. Springer Berlin / Heidelberg, 5201 edition, 2008.

[12] G Bierman, M Hicks, P Sewell, and G Stoyle. Formalizing dynamic software updating. *In Proceedings of USE*, 2003.

[13] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Dynamic software updating, 2001.

[14] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing Dynamic Software Updating. *Update*, pages 1–17, 2003.

[15] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20:10–19, 1987.

[16] Jérémy Buisson and Fabien Dagnat. Introspecting continuations in order to update active code. pages 1–5, Nashville, Tennessee, 2008. ACM.

[17] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *ACM SIGPLAN Notices*, 42(6):66, June 2007.

[18] Scren Christensen. Bisimulation Equivalence is Decidable for Basic Parallel Processes. *Lecture Notes in Computer Science*, 715(CONCUR'93):143–157, 1993.

[19] Silvia Crafa, Francesco Ranzato, and Francesco Tapparo. Saving Space in a Time Efficient Simulation Algorithm. *2009 Ninth International Conference on Application of Concurrency to System Design*, pages 60–69, July 2009.

[20] Amol Deshpande and Michael Hicks. Toward On-line Schema Evolution for Non-stop Systems, September 2005.

[21] S. Gay, V. Vasconcelos, and A. Ravara. Session types for inter-process communication, 2003.

[22] Simon Gay, Vasco Vasconcelos, Antonio Ravara, and António Ravara. Session Types for Inter-Process Communication, 2003.

[23] Simon Gay and Vasco T. Vasconcelos. Asynchronous Functional Session Types, May 2007.

[24] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient systematic testing for dynamically updatable software. *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*, page 1, 2009.

[25] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

[26] Michael Hicks and Scott M Nettles. *Dynamic Software Updating.* PhD thesis, University of Pennsylvania, 2001.

[27] Kohei Honda. Types for Dyadic Interaction. *Lecture Notes in Computer Science*, 715(CONCUR'93):509–523, 1993.

[28] Kohei Honda, Makoto Kubo, and V Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *In ESOPâ98, volume 1381 of LNCS*, volume 171, pages 122–138, July 1998.

[29] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, January 2008.

[30] Hans Hüttel, Naoki Kobayashi, and Takashi Suto. Undecidable equivalences for basic parallel processesâ. *Information and Computation*, 207(7):812–829, July 2009.

[31] Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis, 2002.

[32] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, March 2005.

[33] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. pages 47–57, San Diego, California, United States, 1988. ACM.

[34] R Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* PhD thesis, Technischen Universitat Munchen, 1998.

[35] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. ESOP '09::316–322, 2009.

[36] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, July 2005.

[37] Iulian Neamtiu and Michael Hicks. Safe and Timely Dynamic Updates for Multi-threaded Programs. *SIGPLAN Not.*, 44(6):13–24, 2009.

[38] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, page 37, 2008.

[39] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. *SIGPLAN Not.*, 41(6):72–83, 2006.

[40] Flemming Nielson and Hanne Riis Nielson. Type and Effect Systems. pages 114–136. Springer-Verlag, 1999.

[41] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1 edition, February 2002.

[42] G. D. Plotkin. A Structural Approach to Operational Semantics, 1981.

[43] António Ravara, Vasco T. Vasconcelos, and Simon J. Gay. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1-2):64–87, 2006.

[44] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis. *ACM SIGPLAN Notices*, 40(1):183–194, January 2005.

[45] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. pages 183–194, 2005.

[46] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. pages 398–413, 1994.

[47] C Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[48] Nobuko Yoshida. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited : Two Systems for Higher-Order Session Communication The Honda-Vasconcelos-Kubo Session Typing Sys-. *Electronic Notes in Theoretical Computer Science*, pages 1–20, 2006.

Appendix: Policies

.1 Trace Definitions

$$\frac{\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv U}{U \vdash \mathsf{Exp}(U, \Delta, \Gamma)}\ (\text{:TUsage}) \qquad \frac{\exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P}{\underline{\mathsf{h}} \vdash \mathsf{Exp}(P, \Delta, \Gamma)}\ (\text{:TParTerm})$$

$$\frac{\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv \nu x.P \qquad U \vdash \mathsf{Exp}(P, \Delta, \Gamma)}{\nu x.U \vdash \mathsf{Exp}(\nu x.P, \Delta, \Gamma)}\ (\text{:TParName})$$

$$\frac{\begin{array}{c}\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P_1 \| P_2 \qquad U_1 \vdash \mathsf{Exp}(P_1, \Delta, \Gamma) \qquad U_2 \vdash \mathsf{Exp}(P_2, \Delta, \Gamma) \\ U_1' \vdash \mathsf{next}(U_1, \emptyset, P_2, \Delta, \Gamma, \emptyset) \qquad U_2' \vdash \mathsf{next}(U_2, \emptyset, P_1, \Delta, \Gamma, \emptyset)\end{array}}{U_1' + U_2' \vdash \mathsf{Exp}(P_1 \| P_2, \Delta, \Gamma)}\ (\text{:TPar})$$

$$\frac{U \vdash \mathsf{Exp}(P \| U', \Delta, (\Gamma, \Sigma))}{U \vdash \mathsf{next}(\epsilon, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TEmpt}) \qquad \frac{U \vdash \mathsf{Exp}(P \| \Delta(\underline{\mathsf{h}}); U', \Delta, (\Gamma, \Sigma))}{U \vdash \mathsf{next}(\underline{\mathsf{h}}, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TRecVar})$$

$$\frac{U \vdash \mathsf{Exp}(P \| U', \Delta, (\Gamma, \Sigma))}{\alpha(T); U \vdash \mathsf{next}(\alpha(T), U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TAction}) \qquad \frac{U \vdash \mathsf{next}(U_1, U_2; U', P, \Delta, \Gamma, \Sigma)}{U \vdash \mathsf{next}(U_1; U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TSeq})$$

$$\frac{U_i' \vdash \mathsf{next}(U_i, U', P, \Delta, \Gamma, \Sigma) \qquad i \in 1,2}{U_1' + U_2' \vdash \mathsf{next}(U_1 + U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TIntChoice}) \qquad \frac{U_i' \vdash \mathsf{next}(U_i, U', P, \Delta, \Gamma, \Sigma) \qquad i \in 1,2}{U_1' \& U_2' \vdash \mathsf{next}(U_1 \& U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TExtC}$$

$$\frac{U \vdash \mathsf{next}(U'', U', P, \Delta, \Gamma, \Sigma)}{\nu x.U \vdash \mathsf{next}(\nu x.U'', U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TName})$$

$$\frac{U \vdash \mathsf{next}(U'', U', P, \Delta[\underline{\mathsf{h}} \mapsto \mu\underline{\mathsf{h}}.U''], \Gamma, \Sigma[\underline{\mathsf{h}}' \mapsto P \| \mu\underline{\mathsf{h}}.U''; U']) \qquad \underline{\mathsf{h}}' \text{ free in } \Delta, \Gamma, \Sigma}{\mu\underline{\mathsf{h}}'.U \vdash \mathsf{next}(\mu\underline{\mathsf{h}}.U'', U', P, \Delta, \Gamma, \Sigma)}\ (\text{:TRecDef})$$

.2 Interleaving Definitions

$$\frac{V \vdash \mathsf{intSingle}(U) \qquad \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = U}{V \vdash \mathsf{Traces}(U, \Delta, \Gamma)}\ (\text{:IUsage}) \qquad \frac{\exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P}{\{\underline{\mathsf{h}}\} \vdash \mathsf{Traces}(P, \Delta, \Gamma)}\ (\text{:IParTerm})$$

$$\frac{\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = \nu x.P \qquad V \vdash \mathsf{Traces}(P, \Delta, \Gamma)}{\{\nu x.U | U \in V\} \vdash \mathsf{Traces}(\nu x.P, \Delta, \Gamma)}\ (\text{:IParName})$$

$$\frac{\begin{array}{c}\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) = P_1 \| P_2 \qquad V_1 \vdash \mathsf{Traces}(P_1, \Delta, \Gamma) \qquad V_2 \vdash \mathsf{Traces}(P_2, \Delta, \Gamma) \\ V_1' \vdash \mathsf{intSubSet}(V_1, \emptyset, P_2, \Delta, \Gamma, \emptyset) \qquad V_2' \vdash \mathsf{intSubSet}(V_2, \emptyset, P_1, \Delta, \Gamma, \emptyset)\end{array}}{V_1' \cup V_2' \vdash \mathsf{Traces}(P_1 \| P_2, \Delta, \Gamma)}\ (\text{:IPar})$$

$$\frac{V'' \vdash \mathsf{intSub}(U, U', P, \Delta, \Gamma, \Sigma)}{\bigcup_{U \in V, U' \in V'} V'' \vdash \mathsf{intSubSet}(V, V', P, \Delta, \Gamma, \Sigma)}\ (\text{:ISubSet})$$

$$\frac{V \vdash \mathsf{Traces}(P \| U', \Delta, (\Gamma, \Sigma))}{V \vdash \mathsf{intSub}(\epsilon, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IEmpt}) \qquad \frac{V \vdash \mathsf{Traces}(P \| \Delta(\underline{\mathsf{h}}); U', \Delta, (\Gamma, \Sigma))}{V \vdash \mathsf{intSub}(\underline{\mathsf{h}}, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IRecVar})$$

$$\frac{V \vdash \mathsf{Traces}(P \| U', \Delta, (\Gamma, \Sigma))}{\{\alpha(T); U | U \in V\} \vdash \mathsf{intSub}(\alpha(T), U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IAction}) \qquad \frac{V \vdash \mathsf{intSub}(U_1, U_2; U', P, \Delta, \Gamma, \Sigma)}{V \vdash \mathsf{intSub}(U_1; U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:ISeq})$$

$$\frac{V_i' \vdash \mathsf{intSub}(U_i, U', P, \Delta, \Gamma, \Sigma) \qquad i \in 1, 2}{V_1' \cup V_2' \vdash \mathsf{intSub}(U_1 + U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IIntChoice})$$

$$\frac{V_i' \vdash \mathsf{intSub}(U_i, U', P, \Delta, \Gamma, \Sigma) \qquad i \in 1, 2}{V_1' \cup V_2' \vdash \mathsf{intSub}(U_1 \& U_2, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IExtChoice})$$

$$\frac{V \vdash \mathsf{intSub}(U, U', P, \Delta, \Gamma, \Sigma)}{\{\nu x.U'' | U'' \in V\} \vdash \mathsf{intSub}(\nu x.U, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IName})$$

$$\frac{V \vdash \mathsf{intSub}(U, U', P, \Delta[\underline{\mathsf{h}} \mapsto \mu\underline{\mathsf{h}}.U], \Gamma, \Sigma[\underline{\mathsf{h}}' \mapsto P \| \mu\underline{\mathsf{h}}.U; U']) \qquad \underline{\mathsf{h}}' \text{ free in } \Delta, \Gamma, \Sigma}{\{\mu\underline{\mathsf{h}}.U'' | U'' \in V\} \vdash \mathsf{intSub}(\mu\underline{\mathsf{h}}.U, U', P, \Delta, \Gamma, \Sigma)}\ (\text{:IRecDef})$$

$$\frac{}{\{\epsilon\} \vdash \mathsf{intSingle}(\epsilon)}\ (\text{:ISEmpt}) \qquad \frac{}{\{\alpha(T)\} \vdash \mathsf{intSingle}(\alpha(T))}\ (\text{:ISAction})$$

$$\frac{}{\{\underline{\mathsf{h}}\} \vdash \mathsf{intSingle}(\underline{\mathsf{h}})}\ (\text{:ISAction}) \qquad \frac{V_i \vdash \mathsf{intSingle}(U_i) \qquad i \in 1, 2}{\{U_1'; U_2' | U_i' \in V_i\} \vdash \mathsf{intSingle}(U_1; U_2)}\ (\text{:ISSeq})$$

$$\frac{V_i \vdash \mathsf{intSingle}(U_i) \qquad i \in 1, 2}{V_1 \cup V_2 \vdash \mathsf{intSingle}(U_1 + U_2)}\ (\text{:ISIntChoice}) \qquad \frac{V_i \vdash \mathsf{intSingle}(U_i) \qquad i \in 1, 2}{V_1 \cup V_2 \vdash \mathsf{intSingle}(U_1 \& U_2)}\ (\text{:ISExtChoice})$$

$$\frac{V \vdash \mathsf{intSingle}(U)}{\{\mu\underline{\mathsf{h}}.U' | U' \in V\} \vdash \mathsf{intSingle}(\mu\underline{\mathsf{h}}.U)}\ (\text{:ISRecDef}) \qquad \frac{V \vdash \mathsf{intSingle}(U)}{\{\nu x.U' | U' \in V\} \vdash \mathsf{intSingle}(\nu x.U)}\ (\text{:ISNewName})$$

.3 Degree Definitions

$$\mathsf{d}(P,\Delta,\Gamma) = \begin{cases} 0 & \exists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = P \\ \mathsf{d}(U) & P = U \neq \underline{\mathsf{h}} \wedge \nexists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = P \\ \mathsf{d}(\Delta(\underline{\mathsf{h}})) & P = \underline{\mathsf{h}} \wedge \nexists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = P \\ 1 + \mathsf{d}(P',\Delta,\Gamma) & P = \nu x.P' \wedge \nexists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = P \\ \mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1,\Delta,\Gamma),\mathsf{d}(P_2,\Delta,\Gamma)) & P = P_1 \| P_2 \wedge \nexists \underline{\mathsf{h}}'.\Gamma(\underline{\mathsf{h}}') = P \end{cases}$$

$$\mathsf{d}(U) = \begin{cases} 0 & U = \epsilon \vee U = \underline{\mathsf{h}} \\ 1 & U = \alpha(T) \\ \mathsf{d}(U_1) + \mathsf{d}(U_2) & U = U_1 ; U_2 \\ 1 + \mathsf{max}(\mathsf{d}(U_1),\mathsf{d}(U_2)) & U = U_1 ; U_2 \\ 1 + \mathsf{d}(U') & U = \nu x.U' \vee U = \mu \underline{\mathsf{h}}.U' \end{cases}$$

$$\mathsf{d}^{\mathsf{n}}(n_1,n_2) = \begin{cases} n_1 & n_2 = 0 \\ n_2 & n_1 = 0 \\ 3 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2), \mathsf{d}^{\mathsf{n}}(n_1, n_2 - 1)) & n_1, n_2 > 0 \end{cases}$$

.4 Properties and Proofs

Lemma .1. ***Left and right strictness*** $\mathsf{d^n}(n_1+1, n_2) > \mathsf{d^n}(n_1, n_2)$

Proof:

Case $n_2 = 0$

$n_1 + 1 > n_1$

Case $n_1 = 0$

$\mathsf{d^n}(0+1, n_2) = 3 + \mathsf{max}(\mathsf{d^n}(0, n_2), \mathsf{d^n}(1, n_2 - 1))$

$\mathsf{d^n}(0, n_2) = n_2$

Case $\mathsf{d^n}(0, n_2) > \mathsf{d^n}(1, n_2 - 1)$

$\mathsf{d^n}(1, n_2) = 3 + \mathsf{d^n}(0, n_2) > \mathsf{d^n}(0, n_2)$

Case $\mathsf{d^n}(0, n_2) < \mathsf{d^n}(1, n_2 - 1)$

$\mathsf{d^n}(1, n_2) = 3 + \mathsf{d^n}(1, n_2 - 1) > \mathsf{d^n}(0, n_2)$

Case $n_1, n_2 > 0$

$\mathsf{d^n}(n_1+1, n_2) = 3 + \mathsf{max}(\mathsf{d^n}(n_1, n_2), \mathsf{d^n}(n_1+1, n_2 - 1))$

Case $\mathsf{d^n}(n_1, n_2) > \mathsf{d^n}(n_1+1, n_2 - 1)$

$$\begin{aligned} \mathsf{d^n}(n_1+1, n_2) &= 3 + \mathsf{d^n}(n_1, n_2) \\ &> \mathsf{d^n}(n_1, n_2) \end{aligned}$$

Case $\mathsf{d^n}(n_1, n_2) < \mathsf{d^n}(n_1+1, n_2 - 1)$

$$\begin{aligned} \mathsf{d^n}(n_1+1, n_2) &= 3 + \mathsf{d^n}(n_1+1, n_2 - 1) \\ &> 3 + \mathsf{d^n}(n_1, n_2) \end{aligned}$$

Right strictness follows similarly.

Corollary .2. *If* $n_1 > n_2$ *where* $n_1, n_2 > 0$ *then* $\mathsf{d^n}(n_1, n_2) = 3 + \mathsf{d^n}(n_1 - 1, n_2)$

Lemma .3. $1 + \mathsf{d}^{\mathsf{n}}(n_1, n_2) < 3 + \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2)$

Proof:

Case $n_2 = 0$

$$1 + \mathsf{d}^{\mathsf{n}}(n_1, n_2) = 1 + n_1$$
$$3 + \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2) = 3 + n_1$$
$$1 + n_1 < 3 + n_1$$

Case $n_1 = 0$

$$1 + \mathsf{d}^{\mathsf{n}}(n_1, n_2) = 1 + n_2$$
$$3 + \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2) = 3 + n_2$$
$$1 + n_2 < 3 + n_2$$

Case $n_1, n_2 > 0$

Case $n_1 > n_2$

$$\begin{aligned} 1 + \mathsf{d}^{\mathsf{n}}(n_1, n_2) &= 1 + (3 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2), \mathsf{d}^{\mathsf{n}}(n_1, n_2 - 1))) \\ &= 1 + (3 + \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2)) \quad \text{by corollary .2} \end{aligned}$$

$$3 + \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2) = 3 + (3 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1 - 2, n_2), \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2 - 1)))$$

irrespective of whether $\mathsf{d}^{\mathsf{n}}(n_1 - 2, n_2) > \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2 - 1)$ or $\mathsf{d}^{\mathsf{n}}(n_1 - 2, n_2), \mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2 - 1)$, both are structurally smaller than the current case and hence: $1+(3+\mathsf{d}^{\mathsf{n}}(n_1-1, n_2)) < 3+(3+\mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1-2, n_2), \mathsf{d}^{\mathsf{n}}(n_1-1, n_2 - 1)))$.

Case $n_1 > n_2$

similar to case $n_1, n_2 > 0$

Corollary .4. $n_3 < n_1 \Rightarrow 1 + \mathsf{d}^{\mathsf{n}}(n_1, n_2) < 3 + \mathsf{d}^{\mathsf{n}}(n_3 - 1, n_2)$

Lemma .5. $P \not\equiv U \wedge U' \vdash \mathsf{Exp}(P, \Delta, \Gamma) \Rightarrow \mathsf{d}(U', \Delta, \Gamma) < \mathsf{d}(P, \Delta, \Gamma)$

Proof: by induction over the structure of P.

Case (TParTerm)

$$\frac{\exists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P}{\underline{\mathsf{h}} \vdash \mathsf{Exp}(P, \Delta, \Gamma)}\ (\text{:}\textsc{TParTerm})$$

$\mathsf{d}(\underline{\mathsf{h}}, \Delta, \Gamma) = 0$

$\mathsf{d}(P, \Delta, \Gamma)$ of all $P \not\equiv$ are all > 0.

Case (TParName)

$$\frac{\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv \nu x.P \qquad U \vdash \mathsf{Exp}(P, \Delta, \Gamma)}{\nu x.U \vdash \mathsf{Exp}(\nu x.P, \Delta, \Gamma)}\ (\text{:}\textsc{TParName})$$

by induction, if $P' \not\equiv U'$ then $\mathsf{d}(\nu x.U, \Delta, \Gamma) < \mathsf{d}(P', \Delta, \Gamma)$

$\mathsf{d}(P, \Delta, \Gamma) = 1 + \mathsf{d}(P', \Delta, \Gamma)$

hence $1 + \mathsf{d}(\nu x.U, \Delta, \Gamma) < 1 + \mathsf{d}(P', \Delta, \Gamma)$

Case (TPar)

$$\frac{\begin{array}{c}\nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P_1 \| P_2 \qquad U_1 \vdash \mathsf{Exp}(P_1, \Delta, \Gamma) \qquad U_2 \vdash \mathsf{Exp}(P_2, \Delta, \Gamma) \\ U_1' \vdash \mathsf{next}(U_1, \emptyset, P_2, \Delta, \Gamma, \emptyset) \qquad U_2' \vdash \mathsf{next}(U_2, \emptyset, P_1, \Delta, \Gamma, \emptyset)\end{array}}{U_1' + U_2' \vdash \mathsf{Exp}(P_1 \| P_2, \Delta, \Gamma)}\ (\text{:}\textsc{TPar})$$

$\mathsf{d}(U_1' + U_2', \Delta, \Gamma) = 1 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma)), \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_2, \Delta, \Gamma), \mathsf{d}(P_1, \Delta, \Gamma)))$

where $U_i \vdash \mathsf{Exp}(P_i, \Delta, \Gamma)$

$$\begin{aligned}\mathsf{d}(P_1 \| P_2, \Delta, \Gamma) &= \mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma)) \\ &= 3 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2), \mathsf{d}^{\mathsf{n}}(n_1, n_2 - 1)) \text{ where } n_i = \mathsf{d}(P_i, \Delta, \Gamma)\end{aligned}$$

We need to show:

$$\begin{aligned}&1 + \mathsf{max}(\mathsf{d}(\mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma))), \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_2, \Delta, \Gamma), \mathsf{d}(P_1, \Delta, \Gamma))) < \\ &\quad 3 + \mathsf{max}(\mathsf{d}^{\mathsf{n}}(n_1 - 1, n_2), \mathsf{d}^{\mathsf{n}}(n_1, n_2 - 1))\end{aligned}$$

Case $\mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma)) > \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_2, \Delta, \Gamma), \mathsf{d}(P_1, \Delta, \Gamma)) \wedge$
$\mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1, \Delta, \Gamma) - 1, \mathsf{d}(P_2, \Delta, \Gamma)) > \mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma) - 1)$

Case $P_1 \equiv U_1$

$\mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1, \Delta, \Gamma) - 1, \mathsf{d}(P_2, \Delta, \Gamma)) = \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma) - 1, \mathsf{d}(P_2, \Delta, \Gamma))$

using TUsage. Hence by Lemma .3 we have

$$1 + \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma), \mathsf{d}(P_2, \Delta, \Gamma)) < 3 + \mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1, \Delta, \Gamma) - 1, \mathsf{d}(P_2, \Delta, \Gamma))$$

Case $P_1 \not\equiv U_1$

By induction $U_1 \vdash \mathsf{Exp}(P_1, \Delta, \Gamma) \Rightarrow \mathsf{d}(U_1, \Delta, \Gamma) < \mathsf{d}(P_1, \Delta, \Gamma)$

hence by Corollary .4

$$1+\mathsf{d}(\mathsf{d}^{\mathsf{n}}(\mathsf{d}(U_1,\Delta,\Gamma),\mathsf{d}(P_2,\Delta,\Gamma))) < 3+\mathsf{d}(\mathsf{d}^{\mathsf{n}}(\mathsf{d}(P_1,\Delta,\Gamma)-1,\mathsf{d}(P_2,\Delta,\Gamma)))$$

Other cases with different maximum values are similar.

We represent: $\mathsf{Exp}(P, \Delta, \Gamma)$ as $\Vdash_{\Delta,\Gamma} P$ and $\mathsf{next}(U, U', P, \Delta, \Gamma, \Sigma)$ as $\Vdash_{\Delta,(\Gamma,\Sigma)} P \| U; U'$.

Lemma .6. *If* $\mathsf{Exp}(P', \Delta', \Gamma')$ *succeeds* $\mathsf{Exp}(P, \Delta, \Gamma)$*,* $P \not\equiv \underline{h} \| P'$*, and* $P \not\equiv U_1 \| U_2$ *then* $\mathsf{d}(P', \Delta', \Gamma') < \mathsf{d}(P, \Delta, \Gamma)$*.*

Proof: by inspection of the tableau rules and by the definition of degree. This is straightforward apart from TPar:

Case TPar

$$\frac{\begin{array}{ccc} \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P_1 \| P_2 & U_1 \vdash \mathsf{Exp}(P_1, \Delta, \Gamma) & U_2 \vdash \mathsf{Exp}(P_2, \Delta, \Gamma) \\ \multicolumn{3}{c}{U_1' \vdash \mathsf{next}(U_1, \emptyset, P_2, \Delta, \Gamma, \emptyset) \qquad U_2' \vdash \mathsf{next}(U_2, \emptyset, P_1, \Delta, \Gamma, \emptyset)} \end{array}}{U_1' + U_2' \vdash \mathsf{Exp}(P_1 \| P_2, \Delta, \Gamma)} \ (\text{:TPar})$$

We can prove this straightforwardly apart from when $P_1 \equiv U_1$ or $P_2 \equiv U_2$. In the first case $U_2 \vdash \mathsf{Exp}(P_2, \Delta, \Gamma) \Rightarrow \mathsf{d}(U_2, \Delta, \Gamma) < \mathsf{d}(P_2, \Delta, \Gamma)$ by Lemma .5. The second case follows similarly.

Definition .7. $\Vdash_{\Delta',\Gamma'} P'\|\underline{\mathsf{h}}';U'$ C-succeeds $\Vdash_{\Delta,\Gamma} P\|\underline{\mathsf{h}};U$ if there is a sequence of sequents succeeding each other so that no intermediary sequent is of the form $\Vdash_{\Delta'',\Gamma''} P''\|\underline{\mathsf{h}}'';U''$.

Lemma .8. *If $\Vdash_{\Delta',\Gamma'} P'\|\underline{h}';U'$ C-succeeds $\Vdash_{\Delta,\Gamma} P\|\underline{h};U$ then either:*

1. *$\exists \underline{h}''$ such that $\Gamma(\underline{h}'') \equiv P'\|\Delta(\underline{h}');U'$ or*
2. *$\underline{h}' \in \mathsf{fv}(P\|\Delta(\underline{h});U) \cup \{\underline{h}\}$ or*
3. *$\mathsf{d}(P'\|\underline{h}';U',\Delta',\Gamma') < \mathsf{d}(P\|\underline{h};U,\Delta,\Gamma)$ and $\mathsf{fv}(P'\|\Delta(\underline{h}');U') \subseteq \mathsf{fv}(P\|\Delta(\underline{h});U) \cup \{\underline{h}\}$*

Proof:

If $\underline{\mathsf{h}}' = \underline{\mathsf{h}}$, $U' = U$, and $P' = P$, then using the TRecDef rule we will have created a $\underline{\mathsf{h}}'' \in \mathsf{dom}(\Gamma')$ such that $\Gamma(\underline{\mathsf{h}}'') \equiv P'\|\Delta(\underline{\mathsf{h}}');U'$.

Otherwise, suppose $\Delta(\underline{\mathsf{h}}) = \mu\underline{\mathsf{h}}.U_1$. Either $P'\|\Delta(\underline{\mathsf{h}}');U'$ is a subformula of $P\|\Delta(\underline{\mathsf{h}});U$, or $\underline{\mathsf{h}}'$ is introduced as $\mu\underline{\mathsf{h}}'.U_2$. In the first case $\underline{\mathsf{h}}' \in \mathsf{fv}(P\|\Delta(\underline{\mathsf{h}});U) \cup \{\underline{\mathsf{h}}\}$. In the second case $\mu\underline{\mathsf{h}}'.U_2$ is a subformula of $P\|U_1;U$, in which case $\mathsf{d}(P'\|\underline{\mathsf{h}}';U',\Delta',\Gamma') < \mathsf{d}(P\|\underline{\mathsf{h}};U,\Delta,\Gamma)$ and $\mathsf{fv}(P'\|\Delta(\underline{\mathsf{h}}');U') \subseteq \mathsf{fv}(P\|\Delta(\underline{\mathsf{h}});U) \cup \{\underline{\mathsf{h}}\}$.

Lemma .9. *If* Δ *is a prefix of* Δ' *and* $\mathsf{fv}(P) \subseteq \mathsf{dom}(\Delta)$ *then* $\mathsf{d}(\Delta, \Gamma, P) = \mathsf{d}(\Delta', \Gamma, P)$.

Theorem .10. *Each* trace *tableau is finite.*

Proof:

Assume an infinite tableau τ for parallel usage P. As τ is finite branching, and P contains only finitely many single-threaded usages U in parallel, there must be an infinite path π through τ.

Since $\mathsf{d}(P, \Delta, \Gamma)$, $\mathsf{d}(U)$, and $\mathsf{d}^{\mathsf{n}}(n_1, n_2)$ are all positive descending measures, for all P except those with leading constants, then by Lemma .8 part 3 then there must be an infinite C path in π, $\pi_c = <\Vdash_{\Delta_i,\Gamma_i} P_i \| \underline{\mathsf{h}}_i; U_i>$. As the model is finite, no one constant can occur infinitely often.

For the first $\underline{\mathsf{h}}$ in the path, $\underline{\mathsf{h}}_0$, $\mathsf{fv}(\Delta_0) = \{\underline{\mathsf{h}}\}$, as we have unwrapped no other definitions. The following $\underline{\mathsf{h}}_1$ can then be one of the cases, by Lemma .8:

1. $\Delta(\underline{\mathsf{h}}_1) = \mu \underline{\mathsf{h}}_1.U_1' \wedge \Gamma(\underline{\mathsf{h}}\_1) = P_1 \| U_1'; U_1$, in which case we have a contradiction as this terminates using the TTerm rule.
2. $\underline{\mathsf{h}}_1 \in \mathsf{fv}(P_0 \| \Delta_0(\underline{\mathsf{h}}_0); U_0) \cup \{\underline{\mathsf{h}}_0\}$. As we have introduced no other bindings in Δ_0 then $\mathsf{fv}(\Delta_0) = \{\underline{\mathsf{h}}\}$, and hence $\underline{\mathsf{h}}_1 = \underline{\mathsf{h}}_0$
3. $\mathsf{d}(P_1 \| \underline{\mathsf{h}}_1; U_1, \Delta_1, \Gamma_1) < \mathsf{d}(P_0 \| \underline{\mathsf{h}}_0; U_0, \Delta_0, \Gamma_0)$ and $\mathsf{fv}(P_1 \| \Delta(\underline{\mathsf{h}}_1); U_1) \subseteq \mathsf{fv}(P_0 \| \Delta_0(\underline{\mathsf{h}}_0); U_0) \cup \{\underline{\mathsf{h}}_0\}$

We can find a chain of $\underline{\mathsf{h}}_i$ using the second case. Consider some $\underline{\mathsf{h}}_{i_0}$ is the last $\underline{\mathsf{h}}$ in this chain. Then, using the third case:

$\mathsf{d}(P_{i_{i_0}+1} \| \underline{\mathsf{h}}_{i_{i_0}+1}; U_{i_{i_0}+1}, \Delta_{i_{i_0}+1}, \Gamma_{i_{i_0}+1}) < \mathsf{d}(P_{i_0} \| \underline{\mathsf{h}}_{i_0}; U_{i_0}, \Delta_{i_0}, \Gamma_{i_0})$ and

$\mathsf{fv}(P_{i_{i_0}+1} \| \Delta(\underline{\mathsf{h}}_{i_{i_0}+1}); U_{i_{i_0}+1}) \subseteq \mathsf{fv}(P_{i_0} \| \Delta_{i_0}(\underline{\mathsf{h}}_{i_0}); U_{i_0}) \cup \{\underline{\mathsf{h}}_{i_0}\}$.

By repeating this argument sufficiently often we obtain a contradiction as $\mathsf{d}(P, \Delta, \Gamma)$, $\mathsf{d}(U)$, and $\mathsf{d}^{\mathsf{n}}(n_1, n_2)$ are all non-negative integer measures.

Theorem .11. *Each* int *tableau is finite.*

Proof: similar to Theorem .10.

Lemma .12. $\mathsf{intSubSet}(\mathsf{Traces}(U, \Delta, \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))), \mathsf{Traces}(U', \Delta, \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))), P, \Delta, \Gamma, \Sigma) = \mathsf{intSingle}(\mathsf{next}(U, U', P, \Delta, \Gamma, \Sigma))$

Proof: by induction over the structure of U

Case $U = \underline{\mathsf{h}}$

$$\begin{aligned}
&\mathsf{intSubSet}(\mathsf{Traces}(U, \Delta, \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))), \mathsf{Traces}(U', \Delta, \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))), P, \Delta, \Gamma, \Sigma)\\
&\quad = \mathsf{intSubSet}(\{\underline{\mathsf{h}}\}, \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma)), P, \Delta, \Gamma, \Sigma)\\
&\quad = \textstyle\bigcup_{U'' \in \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))} \mathsf{intSub}(\underline{\mathsf{h}}, U'', P, \Delta, \Gamma, \Sigma)\\
&\quad = \textstyle\bigcup_{U'' \in \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))} \mathsf{Traces}(P \| U'''; U'', \Delta, (\Gamma, \Sigma))
\end{aligned}$$

where $\Delta(\underline{\mathsf{h}}) = \mu\underline{\mathsf{h}}.U'''$

$$\begin{aligned}
&\mathsf{intSingle}(\mathsf{next}(U, U', P, \Delta, \Gamma, \Sigma))\\
&\quad = \mathsf{intSingle}(\mathsf{Exp}(P \| U'''; U', \Delta, (\Gamma, \Sigma)))\\
&\quad = \textstyle\bigcup_{U'' \in \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))} \mathsf{intSingle}(\mathsf{Exp}(P \| U'''; U'', \Delta, (\Gamma, \Sigma)))\\
&\quad = \textstyle\bigcup_{U'' \in \mathsf{Traces}(U', \Delta, (\Gamma, \Sigma))} \mathsf{Traces}(\mathsf{Exp}(P \| U'''; U'', \Delta, (\Gamma, \Sigma)), \Delta, (\Gamma, \Sigma))
\end{aligned}$$

where $\Delta(\underline{\mathsf{h}}) = \mu\underline{\mathsf{h}}.U'''$ then $\mathsf{Traces}(P \| U'''; U'', \Delta, (\Gamma, \Sigma)) = \mathsf{Traces}(\mathsf{Exp}(P \| U'''; U'', \Delta, (\Gamma, \Sigma)), \Delta, (\Gamma, \Sigma))$ by Theorem .13 using lexicographic induction, as $(\Gamma, \Sigma) > \Gamma$.

If $\Sigma \neq \emptyset$ then

Other cases are trivial.

Theorem .13. *If* $V \vdash \mathsf{Traces}(P, \Delta, \Gamma)$ *then* $V \vdash \mathsf{Traces}(\mathsf{Exp}(P, \Delta, \Gamma), \Delta, \Gamma)$.

Proof: by lexicographic induction over (Γ, P), where $\Gamma, \Gamma' < \Gamma$, and simultaneous induction with Lemma .12.

Case $P \equiv P_1 \| P_2 \wedge \nexists \underline{\mathsf{h}}.\Gamma(\underline{\mathsf{h}}) \equiv P$

$$\mathsf{Exp}(P_1, P_2, \Delta)\Gamma \quad = \mathsf{next}(\mathsf{Exp}(P_1, \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset) + \mathsf{next}(\mathsf{Exp}(P_1, \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset)$$

$$\begin{aligned}
\mathsf{Traces}(\mathsf{Exp}(P_1 \| P_2, \Delta, \Gamma), \Delta, \Gamma) \quad &= \mathsf{intSingle}(\mathsf{Exp}(P_1, P_2, \Delta)\Gamma) \\
&= \mathsf{intSingle}(\mathsf{next}(\mathsf{Exp}(P_1, \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset)) \cup \\
&\quad\ \mathsf{intSingle}(\mathsf{next}(\mathsf{Exp}(P_2, \Delta, \Gamma), \emptyset, P_1, \Delta, \Gamma, \emptyset))
\end{aligned}$$

$$\begin{aligned}
\mathsf{Traces}(P_1 \| P_2, \Delta, \Gamma) \quad &= \mathsf{intSubSet}(\mathsf{Traces}(P_1, \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset) \cup \\
&\quad\ \mathsf{intSubSet}(\mathsf{Traces}(P_2, \Delta, \Gamma), \emptyset, P_1, \Delta, \Gamma, \emptyset) \\
&= \mathsf{intSubSet}(\mathsf{Traces}(\mathsf{Exp}(P_1, \Delta, \Gamma), \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset) \cup \\
&\quad\ \mathsf{intSubSet}(\mathsf{Traces}(\mathsf{Exp}(P_1, \Delta, \Gamma), \Delta, \Gamma), \emptyset, P_1, \Delta, \Gamma, \emptyset) \\
&\qquad \text{by the induction hypothesis} \\
&= \mathsf{intSingle}(\mathsf{next}(\mathsf{Exp}(P_1, \Delta, \Gamma), \emptyset, P_2, \Delta, \Gamma, \emptyset)) \cup \\
&\quad\ \mathsf{intSingle}(\mathsf{next}(\mathsf{Exp}(P_2, \Delta, \Gamma), \emptyset, P_1, \Delta, \Gamma, \emptyset)) \\
&\qquad \text{by Lemma .12}
\end{aligned}$$

Lemma .14. *If $A_{\varphi(r,\mathsf{R}(U))}$ simulates σ with relation R then*

$$\forall (q,\sigma) \in R.\, q \not\xrightarrow{L} \;\Rightarrow\; \sigma \not\xrightarrow{L}$$

Proof: contrapositive of $\sigma \xrightarrow{L} \Rightarrow\; q \xrightarrow{L}$, which is easily derivable from the definition of simulation.

Definition .15. $\eta \in U$ if $U, \emptyset \xrightarrow{\eta} \epsilon, R$

Definition .16. $\mathsf{O}^{\sigma}(U) \overset{\text{def}}{=} \{\eta' | \eta \in U \wedge \eta'$ is the largest prefix of η s.t. $\sigma_{init} \xrightarrow{\eta'}{}^{*}\}$

Definition .17. $\Phi(U) \overset{\text{def}}{=} \{A_{\varphi(r_0,\mathsf{R}(U))} | r_0, \varphi \in U\} \cup \{A_{\varphi(\#,\mathsf{R}(U))} | \varphi \in U\}$, where $\mathsf{R}(U)$ comprises of $\#, \_$, and all the static resources occurring in U

Theorem .18. *Modified Theorem 2 from [7]*

1. *An initial usage U is valid iff $\forall \eta' \in \mathsf{O}^{\sigma}(U), \forall A_{\varphi(r,\mathsf{R}(U))} \in \Phi(U)$.*

$$[[\mathsf{B}(\eta')]] \lhd A_{\varphi(r,\mathsf{R}(U))} W A_{\#}$$

2. *The computational complexity of this method is PTIME in the size of U (but U is exponential in the size of the P from which it was generated).*

Theorem .19. *Modified Theorem 2 from [5]*

Let $\Gamma \vdash e : \tau \rhd U$. Then:

1. $\epsilon, R, e, \sigma_{init} \not\to^{*} \eta, R, \mathsf{fail}_K, \sigma$
2. *If U is valid then* $\epsilon, R, e, \sigma_{init} \not\to^{*} \eta, R, \mathsf{fail}_{\varphi(r)}, \sigma$

Definition .20. $\mathsf{O}^{\mathsf{A}_{\varphi(\mathsf{r},\mathsf{R}(\mathsf{U}))}}(U) \overset{\text{def}}{=} \{\eta' | \eta \in U \wedge \eta' \text{ is largest prefix of } \eta \text{ s.t. } q_0 \xrightarrow{\eta'} {}^*\}$

Theorem .21. *For each $A_{\varphi(r,\mathsf{R}(U))} \in \Phi(U)$ where $A_{\varphi(r,\mathsf{R}(U))}$ simulates σ, if:*

$$\forall \eta' \in \mathsf{O}^{\mathsf{A}_{\varphi(\mathsf{r},\mathsf{R}(\mathsf{U}))}}(U).[[\mathsf{B}(\eta')]] \triangleleft A_{\varphi(r,\mathsf{R}(U))} W A_{\#}$$

then

$$\forall \eta' \in \mathsf{O}^{\sigma}(U).[[\mathsf{B}(\eta')]] \triangleleft A_{\varphi(r,\mathsf{R}(U))} W A_{\#}$$

Proof:

As $A_{\varphi(r,\mathsf{R}(U))}$ simulates σ using some relation R' then, by Lemma .14:

$$\forall (q, \sigma) \in R'. \, q \overset{L}{\not\to} \;\Rightarrow\; \sigma \overset{L}{\not\to}$$

Hence the largest prefix reducible in $A_{\varphi(r,\mathsf{R}(U))}$ will conservatively approximate those reducible in σ.

Hence if this larger set of traces is valid then the smaller subset of traces will also be valid.

Note that we can generate some simulation relation using standard techniques such as [19].