Type and Effect Systems for Dynamically Changing Code
DRAFT CORRECTIONS, DO NOT DISTRIBUTE

Gabrielle Anderson
Dependable Systems and Software Engineering
School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ

Abstract

We extend type and effect analyses to permit dynamically changing effects and dynamically changing code in multi-threaded systems with shared resources. We guarantee that after a well typed modification a set of threads will have the specified desired effects and will continue to safely use the resources. We prove the key properties of subject reduction and fidelity (safety) for our general system. We provide an application of our system: dynamic software updating for a multi-threaded asynchronous message passing system. We show how key safety properties from session typing follow from the properties of our general system.

1 Corrections

In Section 4.1 we make explicit how the type judgement for dynamic expressions can be written as either:

$$\varphi \wr \varphi' \wr \Phi \wr \Gamma \vdash e : T \quad \text{or} \quad \xi \wr \Gamma \vdash e : T$$

In Section 4.2 we make explicit the previously implicit assumption that a resource access $\sigma(l)$ always returns a value rather than an arbitrary expression. We also make explicit how the type judgement for fixed expressions can be written as either:

$$\varphi \wr \varphi' \wr \Gamma \vdash e : T \quad \text{or} \quad \xi^s \wr \Gamma \vdash e : T$$

In Section 4.3

$$\begin{aligned}\mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) \quad &\stackrel{\text{def}}{=} \quad \forall \varphi_1'.\varphi_1' \preceq \varphi_1, \forall \varphi_2'.(\varphi_1';(l,T)) \preceq \varphi_2, \\ &\quad \forall \varphi_i \in \mathsf{int}(\varphi_1',\varphi_2').\text{if there is a reduction} \\ &\quad \text{path } \Sigma \xrightarrow{\varphi_i}{}^*\Sigma' \xrightarrow{(l,T)} \Sigma'' \text{ then } \Sigma'(l)=T\end{aligned}$$

should read:

$$\begin{aligned}\mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) \quad &\stackrel{\text{def}}{=} \quad \forall \varphi_1'.\varphi_1' \preceq \varphi_1, \forall \varphi_2'.(\varphi_2';(l,T)) \preceq \varphi_2, \\ &\quad \forall \varphi_i \in \mathsf{int}(\varphi_1',\varphi_2').\text{if there is a reduction} \\ &\quad \text{path } \Sigma \xrightarrow{\varphi_i}{}^*\Sigma' \xrightarrow{(l,T)} \Sigma'' \text{ then } \Sigma'(l)=T\end{aligned}$$

(φ_1' in $(\varphi_1';(l,T)) \preceq \varphi_2$ changed to φ_2').

In Figure 3 include formal definitions of ξ, ξ^s which were previously implicitly used.

In Figure 7

$$\frac{\sigma \vdash \langle e_1 \rangle : S_1 \quad \sigma \vdash \langle e_2 \rangle : S_2 \quad \vdash \sigma : \Sigma \quad \mathsf{compat}(S_1,S_2,\Sigma)}{\sigma \vdash \langle e_1 \rangle \| \langle e_2 \rangle : S_1 \| S_2} \text{ (TPAR)}$$

should read:

$$\frac{\sigma \vdash P_1 : S_1 \quad \sigma \vdash P_2 : S_2 \quad \vdash \sigma : \Sigma \quad \mathsf{compat}(S_1,S_2,\Sigma)}{\sigma \vdash P_1 \| P_2 : S_1 \| S_2} \text{ (TPAR)}$$

($\langle e_1 \rangle, \langle e_2 \rangle$ changed to P_1, P_2).

In Section 5 Fidelity property we include the previously ommited

$$\frac{\Sigma, S_1 \xrightarrow{(l,T)} \Sigma', S_1'}{\Sigma, S_1 \| S_2 \xrightarrow{(l,T)} \Sigma', S_1' \| S_2} \text{ (ABSTRPARONE)}$$

$$\frac{\Sigma, S_2 \xrightarrow{(l,T)} \Sigma', S_2'}{\Sigma, S_1 \| S_2 \xrightarrow{(l,T)} \Sigma', S_1 \| S_2'} \text{ (ABSTRPARTWO)}$$

2 Introduction

Type and effect systems provide a useful abstraction for reasoning about the side effects of programs [22]. Previous work on type and effect systems has focused on expressibility [22] and analyses for specific problems [17, 15, 20], but has assumed that a program's effects would not change at run time.

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 [4]. Commercial products may permit plug-ins 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 [7]. 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 [23, 20, 19, 11]. 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.

Often when making a dynamic modification the programmer may wish to preserve certain behavioural properties after the modification. 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 [17].

Systems can share different types of resources: in a message passing system a message queue may be a shared resource whereas in a shared state system a reference cell may be a shared resource. The type of the value returned when accessing a shared resource can depend on the current state of the resource; for example the type of the value returned by popping an item off a queue depends on the value at the front of the queue. A practical model of effects on shared resources should take account of this dynamic behaviour.

As accesses to shared resources can modify those shared resources and dynamic modifications to code can change the effect of code (the accesses the code performs) the safety of a modification may depend on the state of the shared resources as well as the code performing the accesses [1]. We refer to the current state of the shared resource(s) and the multi-threaded code which has yet to be executed as a *snapshot* of a system.

In this paper we consider the following problem: given a snapshot of a system and some dynamic modifications to the system will the modified system have the desired effect and will it remain type safe? This approach requires that we type a snapshot of a system each time we introduce new dynamic modifications.

The core contributions of this paper are as follows. We provide a type and effect system which tracks changes in effects caused by dynamics modification to expressions. We make use of *world constraints* to track the difference between the effect of a modified expression and its desired effect (which must be specified by the programmer). These constraints provide an abstraction of how the effect of an expression may change. If the constraints are fulfilled we guarantee that after a dynamic modification that the new expression will have the desired effect. We permit the value returned by an effectful primitive acting on shared resources (and its type) to be dependent on the current state of those resources. We guarantee the type of a return value is equal to the locally expected type irrespective of the scheduling of modifications to shared resources made by parallel threads. We prove key properties of our system, namely subject reduction and effect fidelity. We provide an instantiation of our system with rules for asynchronous message passing and a queue to implement communication. We type updates and guarantee that the new communications behaviour will not interfere with any communications protocols, including those which are in progress. This instantiation is the first work on preserving formal behavioural properties besides simple typing for multi-threaded dynamic software update. To the best of our knowledge our general system is the first type and effect system for shared resources where the type of the value returned by accessing shared resources is guaranteed despite its dependency on modifiable shared resources.

The remainder of the paper is structured as follows. In Section 3 we introduce the language and its operational semantics, specifically dynamic modifications to code. In Section 4 we present our type system. We particularly focus on world constraints which permit modification of effects and when it is safe to place threads in parallel which can modify shared resources. In Section 5 we present the subject reduction and safety properties of our system. In Section 6 we provide an application of our system for preserving communications protocols (session typing) in an asynchronous message passing system where the code can be dynamically modified. In Section 7 we present related work. We conclude in Section 8.

3 Language and Operational Semantics

We use a simple lambda-calculus with recursive functions and threads. We include a single effectful primitive $\mathtt{acc}^l$ such as in [15, 22]. The effectful primitive can modify accompanying resources σ. We also include the language construct $\mathtt{dmod}(e)$ which we use to denote where the system may perform dynamically modifications. Expressions and program threads are defined in Figure 1.

The operational semantics of this language, defined in Figure 2, is standard apart from the RAcc and RDyn reductions.

The RAcc reduction modifies the resources according to the resource modification relation $\sigma \xrightarrow{l} \sigma'$. We do not define this function as it will change depending on the situation: in a message passing system it may be a message queue, in a

$$
\begin{array}{rcll}
 & & v & ::= & x \\
e & ::= & v & | & n \\
 & | & \mathtt{rec} f = \lambda x.e & | & b \\
 & | & e\, e & | & () \\
 & | & \mathtt{acc}^l & & \\
 & | & \mathtt{dmod}(e) & P & ::= & \langle e \rangle \\
 & & & | & P \parallel P
\end{array}
$$

Figure 1. Expression and Program Thread Grammars

$$\frac{\sigma,\langle e_1\rangle \longrightarrow \sigma',\langle e_1'\rangle}{\sigma,\langle e_1\, e_2\rangle \longrightarrow \sigma',\langle e_1'\, e_2\rangle}\ (\textsc{RAppOne})$$

$$\frac{\sigma,\langle e_2\rangle \longrightarrow \sigma',\langle e_2'\rangle}{\sigma,\langle v\, e_2\rangle \longrightarrow \sigma',\langle v\, e_2'\rangle}\ (\textsc{RAppTwo})$$

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

$$\frac{\sigma \xrightarrow{l} \sigma'}{\sigma,\langle \mathtt{acc}^l\rangle \longrightarrow \sigma',\langle \sigma(l)\rangle}\ (\textsc{RAcc})$$

$$\frac{}{\sigma,\langle \mathtt{dmod}(e)\rangle \longrightarrow \sigma,\langle \mathsf{DMod}(e,\mu)\rangle}\ (\textsc{RDyn})$$

$$\frac{\sigma,P_1 \longrightarrow \sigma',P_1'}{\sigma,P_1 \parallel P_2 \longrightarrow \sigma',P_1' \parallel P_2}\ (\textsc{RParOne})$$

$$\frac{\sigma,P_2 \longrightarrow \sigma',P_2'}{\sigma,P_1 \parallel P_2 \longrightarrow \sigma',P_1 \parallel P_2'}\ (\textsc{RParTwo})$$

Figure 2. Operational Semantics

shared state system it may be a table implementing references. We do, however, require various properties of this relation which are defined in Section 4.2. The return value of the RAcc reduction is defined as the projection of the resources using the effect. We refer to this projection as a *resource access*.

The RDyn reduction permits us to perform dynamic changes to an expression, using some modification information μ which is supplied at runtime. The auxiliary function DynamicModification (shortened to DMod) can arbitrarily change the expression; DMod can be any dynamic modification function which returns expressions. We require DMod to be a function rather than simply a relation so that we can conservatively estimate the type and effect of the result.

4 Typing

A standard type and effect judgement is:

$$\varphi \wr \Gamma \vdash e : T$$

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. We provide two extensions to this type judgement, one for 'dynamic expressions' in Section 4.1 and another for 'fixed expressions' in Section 4.2. We define typing of threads (which are 'fixed code') in Section 4.3.

4.1 Typing Dynamic Expressions

After dynamically modifying an expression from e to e' we can compare the effect of e' with the effect that we want following the modification. We explain this with an example. Consider the expression:

$$e = \mathtt{acc}^{l_1};\ \mathtt{acc}^{l_2};\ \mathtt{acc}^{l_3}$$

which is modified to the expression:

$$e' = \mathtt{true};\ \mathtt{acc}^{l_2};\ \mathtt{false}$$

We refer to the effect of an expression before a modification as its *prior effect* and the effect of an expression after a modification as its *modified effect*. Consider a situation where we want the modified effect to be equal to the prior effect. To verify that this is the case we can compare the effect of e with the effect of e'. The safety of a modification to an expression depends on the context of that expression. In this example the modification could be safe - the effect of e' is included in the effect of e. If this expression is part of a larger body of code and if, after the modification, the code preceding e' performs an access to label l_1 after performing its prior effect, and the code following e' performs an access to label l_3 before performing its prior effect, then the modification is safe. Hence the modification safety is context dependent.

In order to guarantee safety of this modification we must guarantee the above, namely that the modified code preceding e' performs an access to label l_1 after performing any other accesses and that the modified code following e' performs an access to label l_3 before performing any other accesses. To represent this we define *world constraints*, effects, and types in Figure 3. Effects are a tuple containing the actual effect label l and the return type T which we expect of the resource access $\sigma(l)$. We discuss the significance of the return type in the effect in Section 4.2. Function types have two possible annotations: the first annotation is for a function which is dynamic code and the second annotation is for a function which is fixed code.

$$
\begin{array}{lll}
\varphi & ::= & (l,T) \\
& | & \varphi;\ \varphi \\
\\
S & ::= & \xi^s \\
& | & S \parallel S \\
\\
\phi & ::= & +\varphi \\
& | & -\varphi \\
\\
\Phi & ::= & (\phi \wr \phi') \\
& | & \textsc{Fail} \\
& | & \Phi, \Phi
\end{array}
\qquad
\begin{array}{lll}
\xi & ::= & \varphi \wr \varphi \wr \Phi \\
\\
\xi^s & ::= & \varphi \wr \varphi \\
\\
T & ::= & \textsc{Bool} \\
& | & \textsc{Int} \\
& | & \textsc{Unit} \\
& | & T \xrightarrow{\varphi \wr \varphi' \wr \Phi} T \\
& | & T \xrightarrow{\varphi \wr \varphi'} T
\end{array}
$$

Figure 3. Type, Effect, Effect Bundle, and Future World Constraint Grammars

$$
\mathsf{diff}(\varphi_1,\varphi_2) \stackrel{\text{def}}{=} \begin{cases}
(-\varphi' \wr -\varphi'') & \varphi_1{=}\varphi';\varphi_2;\varphi'' \\
(-\varphi' \wr +\varphi'') & \varphi_1;\varphi''{=}\varphi';\varphi_2 \\
(+\varphi' \wr -\varphi'') & \varphi'';\varphi_1{=}\varphi_2;\varphi' \\
(+\varphi' \wr +\varphi'') & \varphi_2{=}\varphi';\varphi_1;\varphi'' \\
\textsc{Fail} & \text{otherwise}
\end{cases}
$$

Figure 4. Difference function

The constraint $-\varphi$ denotes that the expression doesn't perform effect φ which is expected of it. The constraint $+\varphi$ denotes that the expression performs the effect expected of it and additionally the effect φ. World constraints place restrictions on the effect of the context of the expression. A world constraint $(\phi \wr \phi')$ denotes that an expression places the constraint ϕ on the code preceding it and ϕ' on the code following it. The above modification from e to e' would have the constraint $(-(l_1,T_1) \wr -(l_3,T_3))$.

We generate the constraints using the function difference (shortened to diff), which is defined in Figure 4. This function takes two effects φ_1 and φ_2, where φ_1 is the *desired effect* and φ_2 the actual effect of the modified code. For example, the effect of e is $(l_1,T_1);\ (l_2,T_2);\ (l_3,T_3)$ and the effect of e' is (l_2,T_2). We want to keep the same effect after the modification and hence:

$$
\mathsf{diff}((l_1,T_1);(l_2,T_2);(l_3,T_3),(l_2,T_2)) \quad = \quad (-(l_1,T_1) \wr -(l_3,T_3))
$$

using the first case of the definition of diff.

In order to compare the desired effect to the effect following the modification we need to know this modified effect. Here we determine this using the function EffectDynamicModification (shortened to EffDMod). Similarly to DMod any function which conservatively returns the type and effect of a modified expression is acceptable. We formalise this property as follows:

$$
\mathsf{EffDMod}(e,\mu){=}(\varphi,T) \quad \Rightarrow \quad \varphi \wr \_ \wr \emptyset \vdash \mathsf{DMod}(e,\mu) : T
$$

Up to this point we have assumed that the desired effect would be equal to the prior effect. This may not always be the case, and hence the desired effect is the final component to our type judgement:

$$
\varphi \wr \varphi' \wr \Phi \wr \Gamma \vdash e : T \quad \text{or} \quad \xi \wr \Gamma \vdash e : T
$$

where φ is the effect of e, φ' is the desired effect of the dynamically modified e, and Φ are the world constraints from this expression.

Dynamic modifications can only occur to expressions which are within a `dmod` block. Such expressions are referred to as *dynamic expressions* or *dynamic code*. We define type rules for dynamic expressions in Figure 5. The

$$
\frac{(\textsc{Int},\varphi){=}\mathsf{EffDMod}(n,\mu)}{\emptyset \wr \varphi' \wr \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash n : \textsc{Int}}\ (\textsc{TIntD})
$$

$$
\frac{(\textsc{Bool},\varphi){=}\mathsf{EffDMod}(b,\mu)}{\emptyset \wr \varphi' \wr \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash b : \textsc{Bool}}\ (\textsc{TBoolD})
$$

$$
\frac{x : T \in \Gamma \quad (T,\varphi){=}\mathsf{EffDMod}(x,\mu)}{\emptyset \wr \varphi' \wr \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash x : T}\ (\textsc{TVarD})
$$

$$
\frac{(T,\varphi){=}\mathsf{EffDMod}(\mathtt{acc}^l,\mu)}{(l,T) \wr \varphi' \wr \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash \mathtt{acc}^l : T}\ (\textsc{TAccD})
$$

$$
\frac{\begin{array}{c}
\xi \wr \Gamma, x : T_1, f : T_1 \xrightarrow{\xi} T_2 \vdash e : T_2 \\
(T_1 \xrightarrow{\varphi_2 \wr \varphi_2'} T_2,\varphi){=}\mathsf{EffDMod}(\mathtt{rec} f{=}\lambda x.e,\mu) \\
\xi{=}\varphi_1 \wr \varphi_1' \wr \Phi_1 \quad \xi'{=}\varphi_1 \wr \varphi_1' \wr \Phi_1, \mathsf{diff}(\varphi_1',\varphi_2)
\end{array}}{\emptyset \wr \varphi' \wr \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash \mathtt{rec} f{=}\lambda x.e : T_1 \xrightarrow{\xi'} T_2}\ (\textsc{TLamD})
$$

$$
\frac{\begin{array}{c}
\varphi_1 \wr \varphi_1' \wr \Phi_1 \wr \Gamma \vdash e_1 : T_1 \xrightarrow{\varphi_3 \wr \varphi_3' \wr \Phi_3} T_2 \\
\varphi_2 \wr \varphi_2' \wr \Phi_2 \wr \Gamma \vdash e_2 : T_1 \\
\varphi'{=}\varphi_1';\varphi_2';\varphi_3' \quad (T_2,\varphi){=}\mathsf{EffDMod}(e_1\ e_2,\mu)
\end{array}}{\varphi_1;\varphi_2;\varphi_3 \wr \varphi' \wr \Phi_1 \boxplus \Phi_2 \boxplus \Phi_3, \mathsf{diff}(\varphi',\varphi) \wr \Gamma \vdash e_1\ e_2 : T_2}\ (\textsc{TAppD})
$$

Figure 5. Type Rules For Dynamically Modifiable Code

$$
\begin{array}{lcll}
\phi \oplus \emptyset & = & \phi & \\
-\varphi \oplus +\varphi & = & \emptyset & \emptyset \equiv +\emptyset \equiv -\emptyset \\
\phi_1 \oplus \phi_2 & = & \phi_2 \oplus \phi_1 &
\end{array}
$$

$$
\begin{array}{lll}
\Phi_1 \boxplus \Phi_2 \stackrel{\text{def}}{=} & \{(\phi_1 \wr \phi_2') | (\phi_1 \wr \phi_1') \in \Phi_1, & \\
& (\phi_2 \wr \phi_2') \in \Phi_2, \phi_1' \oplus \phi_2 = \emptyset\} \cup & \\
& \{\textsc{Fail} | (\phi_1 \wr \phi_1') \in \Phi_1, & \\
& (\phi_2 \wr \phi_2') \in \Phi_2, \phi_1' \oplus \phi_2 \neq \emptyset\} &
\end{array}
$$

Figure 6. Combination Operations

$$\frac{}{\emptyset \wr \emptyset \wr \Gamma \vdash n : \text{INT}}\ (\text{TINT})$$

$$\frac{x : T \in \Gamma}{\emptyset \wr \emptyset \wr \Gamma \vdash x : T}\ (\text{TVAR})\varphi$$

$$\frac{}{(l,T) \wr (l,T) \wr \Gamma \vdash \texttt{acc}^l : T}\ (\text{TACC})$$

$$\frac{\xi^s \wr \Gamma, x : T_1, f : T_1 \xrightarrow{\xi^s} T_2 \vdash e : T_2}{\emptyset \wr \emptyset \wr \Gamma \vdash \texttt{rec} f = \lambda x.e : T_1 \xrightarrow{\xi^s} T_2}\ (\text{TLAM})$$

$$\frac{\varphi_1 \wr \varphi_1' \wr \Gamma \vdash e_1 : T_1 \xrightarrow{\varphi_3 \wr \varphi_3'} T_2 \quad \varphi_2 \wr \varphi_2' \wr \Gamma \vdash e_2 : T_1}{\varphi_1 ; \varphi_2 ; \varphi_3 \wr \varphi_1' ; \varphi_2' ; \varphi_3' \wr \Gamma \vdash e_1\ e_2 : T_2}\ (\text{TAPP})$$

$$\frac{\varphi \wr \varphi' \wr \Phi \wr \Gamma \vdash e : T \quad \vdash \Phi \quad \mathsf{fv}(e) = \emptyset}{\varphi \wr \varphi' \wr \Gamma \vdash \texttt{dmod}(e) : T}\ (\text{TMODDEF})$$

$$\vdash \Phi \stackrel{\text{def}}{=} \forall (\phi \wr \phi') \in \Phi . \phi \equiv \phi' \equiv \emptyset \wedge \nexists \text{FAIL} \in \Phi$$

$$\frac{\varphi \wr \varphi' \wr \emptyset \vdash e : T \quad \vdash \sigma : \Sigma \quad \mathsf{compat}((\varphi \wr \varphi'),(\emptyset \wr \emptyset),\sigma) \quad \text{ambient } \mu}{\sigma \vdash \langle e \rangle : \varphi \wr \varphi'}\ (\text{TTHREAD})$$

$$\frac{\sigma \vdash P_1 : S_1 \quad \sigma \vdash P_2 : S_2 \quad \vdash \sigma : \Sigma \quad \mathsf{compat}(S_1, S_2, \Sigma)}{\sigma \vdash P_1 \| P_2 : S_1 \| S_2}\ (\text{TPAR})$$

Figure 7. Type Rules For Fixed Expressions and Threads

main differences between these type rules and standard type and effect systems [22] are world constraints and desired effects.

World constraints are normally generated by taking the difference between the desired effect following the modification and the actual effect following the modification. This can be seen in the TINT, TBOOL, TVAR, TACC, TLAM rules. In TAPP, in addition to taking the difference between the desired effect and the modified effect, we need to combine the world constraints from the sub-expressions. We combine constraints using the operator $\oplus$ and combine world constraints using the operator $\boxplus$; these are defined in Figure 6. We require that the 'future' constraint ϕ_1' from Φ_1 to combine with the 'past' constraint from Φ_2 to result in $\emptyset$. Informally this denotes that any effect that has or hasn't been done by the preceding expression is conversely not done or done by the following expression. If this is not the case we generate FAIL constraints. Note that $\boxplus$ is not commutative.

4.2 Typing Fixed Expressions

An expression which is not within a dmod block cannot be directly modified at run-time. Its sub-expressions, however, may contain a dmod block, and hence the expression is not entirely static. We refer to such expressions as *fixed expressions* (or *fixed code* in the case of threads). Fixed expressions are typed using the judgement:

$$\varphi \wr \varphi' \wr \Gamma \vdash e : T \quad \text{or} \quad \xi^s \wr \Gamma \vdash e : T$$

which denotes that after a possible dynamic modification, which could occur in a sub-expression, the desired effect is φ'. The type rules for fixed expressions are defined in Figure 7.

In non-compound fixed expressions the desired effect after a modification will be the effect before the modification. This is as fixed expressions cannot be modified directly. In TMODDEF we permit the desired effect to be different to the prior effect. This is as the effect of a dmod block is equal to the effect of its subexpression, and its subexpression is a dynamic expression, and hence can be dynamically modified. As we are typing a dynamic sub-expression, it will return world constraints. A FAIL constraint denotes that in some application the world constraints from the sub-expressions do not combine (Figure 6). A world constraint $(\varphi_1 \wr \varphi_2)$, where $\varphi_1 \neq \emptyset$ or $\varphi_2 \neq \emptyset$, denotes that the dynamic expression performs a larger or smaller effect than the desired effect. Hence we define a valid set of world constraints $\vdash \Phi$ as one where the included world constraints are required to be empty and to not be FAIL (Figure 7). This requirement is sufficient to guarantee that the effect of the modified expression is the desired effect. In TMODDEF we require the dynamic sub-expression to be closed to prevent substituting fixed expressions into a dynamic expression.

The return value of the effectful primitive depends on the state of the shared resources (see RACC). As the resources are modifiable at run time, both by the thread in which the access occurs (the *containing thread*) and by parallel threads, it is difficult at the local level to know what the type of the return value will be solely from static information about the resources. Instead we locally infer the type of the return value and record this in the effect; we refer to this type as the *expected type*. When typing the containing thread or the containing thread in parallel with other threads we must verify that the type of the value returned by each resource access is equivalent to its locally inferred expected type, irrespective of resource accesses made by the containing thread or parallel threads.

Shared resources are modified only in the RACC rule. This modification depends on the resources reduction relation. We intentionally do not define this reduction or the structure of the resources, as these will vary depending on the application. Instead we place several requirements on the resources and the resources reduction relation. We use an abstraction of the resources Σ and a reduction relation on abstracted resources. We also place requirements on the abstraction of the resources and the abstracted resources re-

duction relation:

If $\sigma \xrightarrow{l} \sigma'$ then $\sigma(l)$ is defined.
If $\vdash \sigma : \Sigma$ and $\sigma \xrightarrow{l} \sigma'$ then $\Sigma \xrightarrow{(l,\_)} \Sigma' \wedge \vdash \sigma' : \Sigma'$.
If $\vdash \sigma : \Sigma$ and $\sigma(l)$ is defined then $\Sigma(l)$ is defined.
$\sigma(l)$ is always a value.

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 we can abstract the resources, and that a reduction relation exists on resources abstractions which corresponds with the resources reduction relation. The third property requires that the resources abstraction provides simple typing of the return values of resource accesses.

4.3 Typing Threads

When typing an individual thread we take the modification information μ as an ambient to the whole type tree for the body of the thread (TTHREAD). Different threads may have different modification information; intuitively this represents how we may wish to modify different threads in different ways. Hence we can guarantee and reason about the modified effect for different threads. If we do not want to modify a thread we can define the desired effect to be equal to the actual effect and update relation which acts as the identity function.

In order to guarantee simple type safety we still need to verify the expected types. We do this when typing an individual thread or when typing parallel threads. As the return type of a resource access will depend on the state of the shared resources this verification must take account of the starting state of the shared resources. Informally, we require that irrespective of the scheduling of resource accesses , the return type for each access (l, T) is T. When typing a lone thread this simply consists of considering how the effect of the thread before a given access affects the resources. 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. We also require non-interference between the desired effects of the threads.

To state this property formally we define a predicate compatibleGeneral (shortened to compGen) and two auxiliary functions nonInterefere and interleavings (shortened to nonInt and int).

$$\varphi_2 \preceq \varphi_1 \stackrel{\text{def}}{=} (\varphi_1 = \varphi_2 ; \_) \vee (\varphi_1 = \varphi_2)$$

$$\mathsf{compGen}(S_1,S_2,\Sigma) \stackrel{\text{def}}{=} \forall \varphi_1 \wr \varphi_1' \in S_1, \varphi_2 \wr \varphi_2' \in S_2. \; \mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) \wedge \mathsf{nonInt}(\varphi_2,\varphi_1,\Sigma) \wedge \mathsf{nonInt}(\varphi_1',\varphi_2',\Sigma) \wedge \mathsf{nonInt}(\varphi_2',\varphi_1',\Sigma)$$

$$\mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) \stackrel{\text{def}}{=} \forall \varphi_1'.\varphi_1' \preceq \varphi_1, \forall \varphi_2'.(\varphi_2' ; (l,T)) \preceq \varphi_2, \forall \varphi_i \in \mathsf{int}(\varphi_1',\varphi_2').\text{if there is a reduction path } \Sigma \xrightarrow{\varphi_i}{}^{*} \Sigma' \xrightarrow{(l,T)} \Sigma'' \text{ then } \Sigma'(l) = T$$

$$\mathsf{int}(\varphi_1,\varphi_2) \stackrel{\text{def}}{=} \{(l,T) ; \varphi' | \varphi_1 = (l,T) ; \varphi_1' \wedge \varphi' \in \mathsf{int}(\varphi_1',\varphi_2)\} \cup \{(l,T) ; \varphi' | \varphi_2 = (l,T) ; \varphi_2' \wedge \varphi' \in \mathsf{int}(\varphi_1,\varphi_2')\}$$

where $\xrightarrow{\varphi}{}^{*}$ is the reflexive and transitive closure of $\xrightarrow{(l,T)}$.

The function interleavings provides all possible interleavings of two effects. The predicate nonInterefere 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.

The predicate compGen is a formalisation of model checking the possible interleavings of the effects on the abstracted resources Σ. Such an approach has exponential complexity. This is, however, the general solution where the only properties we have of the resources and their reductions are as in Section 4.2. Given more information about the resources reduction relation then we can define a predicate compatible (shortened to compat) such that:

$$\mathsf{compat}(S_1,S_2,\Sigma) \quad \Rightarrow \quad \mathsf{compGen}(S_1,S_2,\Sigma)$$

In the TTHREAD and TPAR rules 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 provide an example of such a definition in Section 6.

5 Properties

We present the main properties of the general system, which are subject reduction and fidelity. These properties hold even in the presence of dynamic modifications to the code and effects.

5.1 Subject Reduction

Informally, subject reduction states that any reduction except dynamic modification preserves or decreases the type and effect of an expression [17], and dynamic modification reductions modify the effect to the desired effect. We refer to this property as *effect reduction validity*. We formalise reduction validity as the smallest transitive relation defined over:

$$\frac{}{\vdash \varphi \rightarrow \varphi} \text{ (ESKIP)} \qquad \frac{}{\vdash (l,T) ; \varphi \rightarrow \varphi} \text{ (EACC)}$$

$$\frac{}{\vdash \varphi_1 \wr \varphi_2 \rightarrow \varphi_2 \wr \varphi_2}\ (\textsc{EDMod})$$

$$\frac{\vdash \varphi_1 \rightarrow \varphi_1' \quad \vdash \varphi_2 \rightarrow \varphi_2'}{\vdash \varphi_1 \wr \varphi_2 \rightarrow \varphi_1' \wr \varphi_2'}\ (\textsc{EThread})$$

$$\frac{\vdash S_1 \rightarrow S_1'}{\vdash S_1 \| S_2 \rightarrow S_1' \| S_2}\ (\textsc{EParOne}) \qquad \frac{\vdash S_2 \rightarrow S_2'}{\vdash S_1 \| S_2 \rightarrow S_1 \| S_2'}\ (\textsc{EParTwo})$$

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. EDMod describes reductions which perform dynamic modifications and denotes that we consider the effect of a thread being modified to its desired effect to be a valid reduction. EAcc describes how the effect of a thread may change, and EParOne and EParOne describe how the effects of parallel threads may change.

Theorem 5.1. ***Subject Reduction***
If $\sigma \vdash P : S$ *and* $\sigma, P \rightarrow \sigma', P'$ *then*

$$\sigma' \vdash P' : S' \qquad \vdash S \rightarrow S'$$

We provide a sketch proof of subject reduction as follows. We prove subject reduction by induction over the operational semantics.

The most important case in the proof for guaranteeing effect reduction validity is modification reductions on a $\texttt{dmod}(e)$ expression. This case is covered as, in the TModDeff rule, we require that the world constraints of the sub-expression to be the null constraint $(\emptyset \wr \emptyset)$ and for there to be no Fail constraints. Hence the actual effect after the modification will always be equal to the desired effect. As we require that, at the thread level, the actual effects and the desired effects of all threads do not interfere with the return types of resource accesses then we will preserve simple typing of resource accesses in a modified thread.

The most important case for guaranteeing simple type safety and 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, we by the compatibility property that the expected types are invariant irrespective of the scheduling of their effects. This is verified by performing model checking. Hence any reductions will not invalidate this property.

5.2 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 [14]. To prove this property we define a reduction relation on resources abstractions and parallel effects which denotes how an effect modifies the abstract resources:

$$\frac{\Sigma \xrightarrow{(l,T)} \Sigma'}{\Sigma,(l,T);\varphi \wr (l,T);\varphi' \xrightarrow{(l,T)} \Sigma',\varphi \wr \varphi'}\ (\textsc{AbstrStateRed})$$

$$\frac{\Sigma, S_1 \xrightarrow{(l,T)} \Sigma', S_1'}{\Sigma, S_1 \| S_2 \xrightarrow{(l,T)} \Sigma', S_1' \| S_2}\ (\textsc{AbstrParOne})$$

$$\frac{\Sigma, S_2 \xrightarrow{(l,T)} \Sigma', S_2'}{\Sigma, S_1 \| S_2 \xrightarrow{(l,T)} \Sigma', S_1 \| S_2'}\ (\textsc{AbstrParTwo})$$

Theorem 5.2. ***Fidelity***
If $\sigma \vdash P : S$ *and* $\sigma, P \xrightarrow{l} \sigma', P'$ *and* $\vdash \sigma : \Sigma$ *then*

$$\Sigma, S \xrightarrow{(l,T)} \Sigma', S' \qquad \vdash \sigma' : \Sigma'$$

This result follows from our properties of the resources and the resources reduction relation in Section 4.2.

6 Application: Session Types and Multi-Threaded Dynamic Software Update

Type and effect systems have been applied to multi-threaded asynchronous message passing system which permit typed communications which are more complex than typed channels which only carry values of a certain type [14, 5, 25, 18, 12, 22]. The principle aim of that work is to provide static typing of communications protocols.

We provide a simplified version of the multi-party session typing [14, 5]. We assume that we have pre-distributed some uni-directional channels c, where one thread holds one end and sends on the channel and the other thread holds the other and receives on the channel. Each thread's effect is specified separately, rather than being specified in a global session type [14, 5]. We provide communications primitives to send and receive values on channels, are syntactic sugar aliasing our $\texttt{acc}^l$ primitives:

$$\mathsf{send}(c,v) \stackrel{\text{def}}{=} \texttt{acc}^{c!\langle v\rangle} \qquad \mathsf{receive}(c) \stackrel{\text{def}}{=} \texttt{acc}^{c?()}$$

We define the structure of the resources, which is a mapping from channels to message queue, and the structure of the resources abstraction, which is a mapping from channels to lists of types of the messages in the queue. We also define the projection of the resources and resources abstractions using an effect label:

$$\begin{array}{rcl@{\qquad}rcl} \sigma & ::= & c \mapsto q & q & ::= & v \\ & | & \sigma,\sigma & & | & q,q \end{array}$$

$$\begin{array}{lll} \Sigma ::= & c \mapsto Q & Q ::= \quad T \\ \mid & \Sigma,\Sigma & \mid \quad Q,Q \end{array}$$

$$\sigma(l) \stackrel{\text{def}}{=} \begin{cases} v & \sigma(c)=v,q \wedge l=c?() \\ () & l=c!\langle v\rangle \end{cases}$$

$$\Sigma(l) \stackrel{\text{def}}{=} \begin{cases} T & \Sigma(c)=T,Q \wedge l=c?() \\ \textsc{Unit} & l=c!\langle v\rangle \end{cases}$$

We define the resources reduction relation and the abstract reduction relation.

$$\frac{\sigma'=\sigma[c \mapsto \sigma(c),v]}{\sigma \xrightarrow{c!\langle v\rangle} \sigma'} \text{ (RSND)} \qquad \frac{\sigma(c)=v,q \quad \sigma'=\sigma[c \mapsto q]}{\sigma \xrightarrow{c?()} \sigma'} \text{ (RRCV)}$$

$$\frac{\Sigma'=\Sigma[c \mapsto \Sigma(c),T] \quad v : T}{\Sigma \xrightarrow{c!\langle v\rangle} \Sigma'} \text{ (RSNDA)}$$

$$\frac{\Sigma(c)=T,Q \quad \Sigma'=\Sigma[c \mapsto Q]}{\Sigma \xrightarrow{c?()} \Sigma'} \text{ (RRCVA)}$$

Finally we define the state typing judgement which relates resources and resources abstractions:

$$\frac{v : T}{\vdash v : T} \text{ (TVAL)} \qquad \frac{\vdash q_1 : Q_1 \quad \vdash q_2 : Q_2}{\vdash q_1,q_2 : Q_1,Q_2} \text{ (TQUEUE)}$$

$$\frac{\vdash q : Q}{\vdash c \mapsto q : c \mapsto Q} \text{ (TMAP)} \qquad \frac{\vdash \sigma_1 : \Sigma_1 \quad \vdash \sigma_2 : \Sigma_2}{\vdash \sigma_1,\sigma_2 : \Sigma_1,\Sigma_2} \text{ (TSTATECOMP)}$$

This system can be used to guarantee that threads communicate according to a communications protocol. We can make use of our definitions and the complementary function [12] to define a linear compatible predicate. The complementary function is defined as:

$$\overline{\varphi} \stackrel{\text{def}}{=} \begin{cases} (c?(),T) & \varphi=(c!\langle v\rangle,\textsc{Unit}) \wedge v : T \\ \overline{\varphi'};\overline{\varphi''} & \varphi=\varphi';\varphi'' \end{cases}$$

The function is intentionally only partially defined, as we make use of this and the fact that channels are unidirectional.

In order to separate the effect of a thread on a specific channel from the rest of the thread's effect we define a monotonic projection function:

$$\varphi \lfloor c \stackrel{\text{def}}{=} \begin{cases} \varphi & \varphi=(c!\langle v\rangle,T) \\ \varphi & \varphi=(c?(),T) \\ \emptyset & \varphi=(c' \_,T) \\ \varphi' \lfloor c;\varphi'' \lfloor c & \varphi=\varphi';\varphi'' \end{cases}$$

We can now define the compatible predicate. We define $\mathsf{dom}(\varphi)$ to be the set of channels which are used by effect labels in φ and make use of an auxiliary relation compatibleSubFunction (shortened to compSub). We use the general definition of the compatible predicate:

$$\begin{array}{rcl} \mathsf{compGen}(S_1,S_2,\Sigma) & \stackrel{\text{def}}{=} & \forall \varphi_1 \wr \varphi_1' \in S_1, \varphi_2 \wr \varphi_2' \in S_2. \\ & & \mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) \wedge \mathsf{nonInt}(\varphi_2,\varphi_1,\Sigma) \wedge \\ & & \mathsf{nonInt}(\varphi_1',\varphi_2',\Sigma) \wedge \mathsf{nonInt}(\varphi_2',\varphi_1',\Sigma) \end{array}$$

but redefine the nonInterefere predicate:

$$\begin{array}{rcl} \mathsf{nonInt}(\varphi_1,\varphi_2,\Sigma) & \stackrel{\text{def}}{=} & \forall c \in (\mathsf{dom}(\varphi_1) \cap \mathsf{dom}(\varphi_2)). \\ & & \varphi_1'=\varphi_1 \lfloor c \wedge \varphi_2'=\varphi_2 \lfloor c \wedge \\ & & \big(\mathsf{nonIntSub}(\varphi_1',\varphi_2',\Sigma,c) \vee \\ & & \mathsf{nonIntSub}(\varphi_2',\varphi_1',\Sigma,c)\big) \end{array}$$

$$\begin{array}{rcl} \mathsf{nonIntSub}(\varphi_1,\varphi_2,\Sigma,c) & \stackrel{\text{def}}{=} & (\varphi_1=\varphi_1';\varphi_1'') \wedge \Sigma \xrightarrow{\varphi_1'}{}^{*} \Sigma' \wedge \\ & & \Sigma'(c)=\emptyset \wedge \varphi_1''=\overline{\varphi_2} \end{array}$$

The nonInterefereSub predicate expresses that there is a reduction path on the resources abstraction which leads to the communications queue for channel c being empty. Informally this corresponds to one thread receiving any messages already in the communications queue. The predicate also expresses that, after emptying the queue, the effect of the receiving thread is complementary to the effect of the sending thread. The nonInterefereSub predicate defines that, for every channel which is shared in the two effects, nonInterefereSub holds for the projection of the effects using that channel. As we do not know which effect represents the sending thread or which represents the receiving thread we require either $\mathsf{nonIntSub}(\varphi_1',\varphi_2',\Sigma,c)$ or $\mathsf{nonIntSub}(\varphi_2',\varphi_1',\Sigma,c)$.

This definition of compatible implies compatibleGeneral. We provide a sketch proof as to why this implication is valid. Given an empty communications queue for a given channel, if the effects on a channel are complementary then, by definition of the complementary function, the types of the messages which the receiver is expecting are the same as the type of the messages the sender will send. In order to guarantee an empty communications queue for a given channel we reduce the abstraction of the current resources using the effect of the receiver until the resources abstraction queue is empty. Hence the type received is invariant irrespective of other threads' effects if this predicate is fulfilled.

This compatibility relation and approach to updating a communications protocol in a multi-threaded message passing system formalises our proposed approach in previous work. In [1] we explain how complementary effects are not sufficient and how the safety of a modification depends on the messages in the communications queue. The standard session typing properties of subject reduction and fidelity [14, 3] follow directly from our generalisation of these properties. Communications safety informally denotes that the values received from a communications queue are of the type which is accepted by the receiver. Hence communications safety is covered by our subject reduction property.

7 Related Work

Type and effect systems are widely used, from resource usage analysis [15] to dynamic software update [20] to se-

cure service composition [2]. The definitive work on type and effect systems is provided in [17, 22]. In [17] the authors provide the contribution type and effect systems for mutable state which guarantees that an expression can only access regions specified by its effect or private regions. This makes use of the novel concept of effect masking, where side effects cannot be seen outside a given expression. In [22] the authors present other applications of type and effect systems, including for communications systems. The authors also discuss subtyping, subeffecting and polymorphism and the methodology of including these properties in a system.

Resource access control for concurrent systems has been approached in various ways. It has been approached for specific problems such as session typing, discussed below. In [16] the authors guarantee that all resource accesses in a concurrent pi-calculus setting occur according the resource usage specification, and that necessary accesses such as the file close operations are eventually performed unless the program diverges. This is enforced by requiring all possible access traces are included in the resource usage specification. The authors present a model checking algorithm and implementation to check this property. In [21] the authors statically determine whether concurrent programs in a simple functional language adhere to an access control. Access policies are defined using finite state automata. The authors reduce the state space search by abstracting away behaviours inside atomic sections and simply check the interleavings of atomic sections. We extend [16, 21] by including a general representation of state and effectful primitives whose return value is defined by the state.

Software extension safety has studied using many methodologies, including using hardware-based protection and paging mechanisms [8], certifications and Hoare-like logic [7], and encapsulation and static type safety [4]. This work has been extended in dynamic software updating (DSU) which aims to dynamically modify code at runtime, preserving various safety properties. Most of the work focuses on preserving simple type safety [6, 23, 20, 19], but also includes preserving results of regression testing [11]. In [6, 23] the authors work in a single-threaded system assuming an update primitive, which is similar to our DMod function. In [20] the authors delineate regions in which an update cannot be visible, and are able to infer regions of safety where updates can occur. The authors make use of these safe regions in [19] to extend the work to multi-threaded systems where they use a "check-in" protocol to only perform an update when all threads are ready to perform it. They also present experimental results of the delay between updates being introduced and when a suitable update point occurs and discuss the balance between safety and timeliness. In both [20] and [19] the authors make use of effect systems which use the effects before and after a given expression. These effects are used for the analysis which is not focused on preserving the effects but on how types are concretely used.

Most session type analyses are based on the early work in [12, 13], though recently some errors in these papers have been discovered and corrected [25]. The early papers on multi-party session types are [14, 3]. The work in [14] presents an asynchronous multi-party session type system which includes a limited form of delegation and a progress property to guarantee that a well typed session will not deadlock, in and of itself. The work in [3] is similar to that of [14] except for its communications are synchronised and hence a guaranteed ordering on communications can be included. Also it places a restriction of not permitting recursive session types. In [5] the authors build on the work in [14] and provide full, transparent, delegation as well as a progress guarantee that different well typed sessions will not interfere with each other and cause deadlock. Most work on session types use π calculus style calculi [12, 18, 5, 25, 14], though some use a λ calculus formulation [9, 10, 24].

8 Conclusions

We introduce world constraints, which abstract the changes required to the effect of code surrounding a modified expression in order for the modification to be safe. We show that fulfilling these constraints is sufficient to guarantee that an expression has the desired effect after a modification. We prove subject reduction and fidelity, which includes proving that the expected type of a given resource access is guaranteed irrespective of the scheduling of resource accesses by parallel threads. We provide an application of our system: performing dynamic software update on a multi-threaded message passing system. We outline how we achieve subject reduction, fidelity and communications safety. This is the first formal approach to preserving behavioural properties in multi-threaded dynamic software update. To the best of our knowledge our general system is the first type and effect system for shared resources where the type of the value returned by accessing shared resources is guaranteed despite its dependency on modifiable shared resources.

Our approach requires guaranteeing non-interference between the effects of parallel threads on the shared resources. We show how, in the general case, this is an exponential problem. When we have an instance of the system, however, this complexity of the problem can be decreased. We provide an example instantiation, where the description of compatibility is based on previous work [1].

We approach the problem by verifying safety of dynamic modifications on a snapshot of the system. The approach of verifying snapshots may be unrealistic in some situations.

Similarly, when we have an instance of the system with certain restrictions on dynamic modifications we may be able to relax the requirement of verifying shapshots of the state and code and be able to verify the system as a whole [1].

Acknowledgements

Thanks to Julian Rathke for suggestions of related work and for proof reading this paper, in various versions. Thanks also for his suggesting to generalise previous attempts at this problem aimed specifically at DSU for message passing systems.

References

[1] Austin Anderson and Julian Rathke. Migrating protocols in multi-threaded message-passing systems. In *Proceedings of the 2nd Workshop on ACM Workshop on Hot Topics in Software Upgrades. Orlando, USA*, 2009.

[2] M. Bartoletti, P. Degano, G.L. Ferrari, and R. Zunino. Secure service orchestration. *Lecture Notes in Computer Science*, 4677:24, 2007.

[3] Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. *Electron. Notes Theor. Comput. Sci.*, 241:3–33, 2009.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[5] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically merged multiparty sessions. Technical report, 2008. `http://www.doc.ic.ac.uk/~yoshida/paper/global_progress.pdf`.

[6] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution. Warsaw, Poland*, 2003.

[7] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 66–77, New York, NY, USA, 2007. ACM.

[8] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 140–153, New York, NY, USA, 1999. ACM.

[9] Simon Gay, V. T. Vasconcelos, and Antonio Ravara. Session types for interprocess communication. Technical Report 133, Department of Computing, University of Glasgow, 2003.

[10] Simon Gay and Vasco T. Vasconcelos. Asynchronous functional session types. Technical Report 2007–251, Department of Computing, University of Glasgow, May 2007.

[11] Christopher Hayden, Eric A Hardisty, Michael Hicks, and Jeffrey Foster. Invited paper: Efficient systematic testing for dynamically updatable software. In *Proceedings of the 2nd Workshop on ACM Workshop on Hot Topics in Software Upgrades. Orlando, USA*, University of Maryland, College Park, USA, 2009.

[12] Kohei Honda. Types for dyadic interaction. In *Lecture Notes in Computer Science, Volume 715/1993, pages 509-523*, 1993.

[13] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP 98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

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

[15] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, March 2005.

[16] Naoki Kobayashi, Kohei Suenaga, and Lucian Wischik. Resource usage analysis for the -calculus. In *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, chapter 20, pages 298–312. 2006.

[17] Lucassen and Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.

[18] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 316–332, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44(6):13–24, 2009.

[20] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1):37–49, 2008.

[21] N. Nguyen and J. Rathke. A typed static analysis for a concurrent policy based resource access control. In *First Program Analysis for Security and Safety Workshop Discussion (PASSWORD 2006), co-located with the Twentieth European Conference on Object-Oriented Programming (ECOOP 2006)*.

[22] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[23] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32 nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 183–194, 2005.

[24] Vasco T. Vasconcelos, Simon J. Gay, and Antonio Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[25] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.