Resource Access with Variably Typed Return

Gabrielle Anderson
University of Southampton
Southampton, Hants, UK
gabrielle.anderson@soton.ac.uk

Julian Rathke
University of Southampton
Southampton, Hants, UK
jr2@ecs.soton.ac.uk

Abstract

Shared resources are a feature of many concurrent distributed systems. Access to these resources often involves using data of different types at each access. We consider the use of static analysis to guarantee type safety in such such systems and provide a view of generalised resource usage which subsumes that of session typing systems.

1 Introduction

In many real world systems we often need to be able to obtain, and use type-safely, values from shared resources; such systems include, for example, multithreaded message passing systems, where the shared resources are channels with associated queues, or those where mobile code is transmitted between locations. When the type or the effect of these values is variable, say for untyped channels in a message passing system, it is possible to send a value which is not of the type that the receiver is expecting. If the receiver does not dynamically check the type of the received value itself it could then use the value inappropriately and cause a run time type error.

Traditionally many systems solve this problem by performing runtime checks on all values received. This is true, for example, for deserialisation of objects in Java. This approach has obvious runtime costs and in general one might seek to mitigate these costs through static analysis. There are different static approaches which can be applied to this problem; we can use model checking to examine the types of all possible values returned, and in some cases we can use structural analyses of the impure actions such as sending messages. In the specific case of message passing systems a number of static analyses known as session typing [2–6] have been developed to address this issue. These approaches leverage the semantics of the message passing system, such as blocking receives and FIFO queues for sent messages, to observe that if two threads act on a channel and their actions are complementary [4] then there will be no run time type errors. We argue that the session typing analysis exemplifies a more general analysis of type and effects for resource accesses with variable types. The specific details of message passing systems can be factored out in to a semantics of resource access so that the underlying mechanics of the type safety proof are independent of the particular resources being accessed. Doing this has the advantages of making the proof of type safety robust with respect to changes in the resource model being used, thereby allowing one to obtain the guarantees of type safety without having to reprove the result.

In this paper, following [7], we present a generalised notion of resource with variably typed access, discuss static analysis for these and state the key properties of subject reduction and fidelity for this analysis. We show how this general model can be instantiated to yield session type analyses for message-passing systems. Crucially we 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 (Section 4). The general solution to proving safety is a model checking approach (for which we do not consider the algorithms). We show how to prove safety in various examples, and how the additional knowledge allows us to simplify and reduce the cost of the proofs.

$$[c \mapsto \emptyset]\, c!\langle \textsc{Bool} \rangle \parallel c?(\textsc{Int}) \\ \to [c \mapsto \textsc{Bool}]\, \varepsilon \parallel c?(\textsc{Int}) \\ \to \texttt{error}$$

Figure 1: Message Passing Error

$$[c \mapsto \textsc{Int}]\, c!\langle \textsc{Bool} \rangle \parallel c?(\textsc{Int}) \\ \to [c \mapsto \textsc{Int}, \textsc{Bool}]\, \varepsilon \parallel c?(\textsc{Int}) \\ \to [c \mapsto \textsc{Bool}]\, \varepsilon \parallel \varepsilon$$

Figure 2: Message Passing Success

$$[c \mapsto \emptyset]\, \varphi \parallel \varphi \parallel c?(\textsc{Int}); c?(\textsc{Bool}) \\ \to \ldots \\ \to [c \mapsto \textsc{Int}, \ldots]\, \varepsilon \parallel \varepsilon \parallel c?(\textsc{Bool}) \\ \to \texttt{error}$$

where $\varphi = c!\langle \textsc{Int} \rangle; c!\langle \textsc{Bool} \rangle$

Figure 3: Message Passing Error From Interleaving

2 Message Passing Systems

In this section we consider message passing systems as a means to discussing our generalised notion of resource. Message passing systems consist of multiple threads of code that uses shared channels. Accesses to shared channels can add and remove values of varying types from channel queues.

In message passing systems type usage errors can occur when the value obtained from a receive action is not of the expected type, for example consider the abstracted code effects in Figure 1 which uses a suggestive notation $c \mapsto T_1, \ldots, T_k$ for the type of a channel queue and $c!\langle T \rangle$ for code which sends a value of type T on channel c and $c?(T)$ for code which expects to receive a value of type T.

Session typing analyses are static analyses that can be used 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. In essence this is done as follows: perform a type and effect analysis on the code to obtain an effect which represents the behaviour of the system on resources and then define some predicate over these effects that exploits knowledge of the semantics of resources to ensure that no type errors occur. For example, for systems in which it is known that only two threads use each channel; one thread for sending only and one for receiving only, then we could define a predicate that asserts that for each receive action in a given thread there is an entry at the front of queue of the correct type or the queue is empty and there is a complementary send of the same type in the other thread. This will then catch the error in Figure 1, as the value sent is not of the type expected by the receiver. Note that whether or not a type error occurs may depends on the starting state of the resources. For example, with the same code and different starting resources, as in Figure 2, no error occurs. The predicate described above accounts for this. If we could not rely on channels only being used by two threads, or that the threads use the channel to send or receive only, then the effects being complementary would not be a sufficient condition to prevent errors; in Figure 3 the effect of each of the senders is complementary to that of the receiver, but as there are multiple senders the sends can interleave resulting in a value of an unexpected type being received.

What we see from the example above is that in checking type safety of code that accesses resources the relevant information is

- the code's effect on the resources,
- the initial state of the resources, and
- semantics of the resources within the system

The first of these can be calculated using a standard type and effect system [8]. The latter two inform our generalised analysis.

3 Generalised System

We now present our generalised resource system and describe the type and effect analysis used to guarantee type safety. We will work with a simply-typed lambda calculus with recursion and primitives $\mathtt{acc}^l$ to describe resource access. The labels l are of the form $\alpha(v_1,\ldots,v_n)$ for some *action* α and values v_i. We will consider systems P of parallel threads of these lambda terms. We parameterise our work on a *resource model*, which is a collection of *states* ranged over by σ and a function that maps a state and a label to a return value and another state: we write $\sigma(l) = v$ and $\sigma \xrightarrow{l} \sigma'$ to denote this map. The semantics of systems is given with respect to a resource state: $[\sigma]P \to [\sigma']P'$ in a mostly obvious way such that a resource access $\mathtt{acc}^l$ respects the semantics of the resource model:

$$\frac{\sigma \xrightarrow{l} \sigma'}{[\sigma]\,\mathtt{acc}^l \to [\sigma']\,\sigma(l)}$$

The resource model is a generalised model, and hence can be instantiated with either finite or infinite models. In order to present our analysis we make use of an *abstract resource model* which abstracts away from particular values associated with resource states and provides a representation of the resource types. An abstract resource model is a collection of abstract states ranged over by Σ and a function that maps a state and an abstract label to a type and another abstract state: we write $\Sigma(L) = T$ and $\Sigma \xrightarrow{L} \Sigma'$ as above. Abstract labels L are of the form $\alpha(T_1,\ldots,T_n)$ so that a label l corresponds to a unique abstract label L via simple typing of values. We relate the resource model and abstract resource model via an abstraction map Λ with the property that $\sigma \xrightarrow{l} \sigma'$ implies $\Lambda(\sigma) \xrightarrow{L} \Lambda(\sigma')$ and $\Gamma \vdash \sigma(l) : \Lambda(\sigma)(L)$, under type variable assumptions Γ.

We would now like to define our type and effect system in a standard way following [8] where the judgements are of the form $\varphi\,;\Gamma \vdash t : T$ for effects φ and simple type T. However, as we are working in a system in which resource accesses may return values of different types depending on the state of the resource it is not generally possible to locally determine a type for the resource access primitive $\mathtt{acc}^l$. Indeed, the type of this expression will depend on the state in which it is accessed. What we can do locally is determine the *expected* type of the $\mathtt{acc}^l$ expression with respect to the remainder of the thread in which the value returned is used. This generates a constraint on the effect for that thread which will be solved globally when all threads are placed together in parallel. In order to do this, the effect of a primitive access $\mathtt{acc}^l$ takes the form (L,T) where L is the corresponding abstract label and T records the expected type of the value to be returned.

$$\frac{}{(L,T)\,;\Gamma \vdash \mathtt{acc}^l : T}$$

Our language of effects is

$$\varphi ::= \varepsilon \mid x \mid (L,T) \mid \varphi;\varphi \mid \mu x.\varphi \qquad\qquad \Phi ::= \varphi \mid \Phi \parallel \Phi$$

where we allow sequencing of effects and recursive effects (infinitely unfolded). Given an effect φ it is easy to see how to lift the reduction relation over states and threads to abstract states and effects $[\Sigma]\,\varphi \to [\Sigma']\,\varphi'$ by making use of the $\Sigma \xrightarrow{L} \Sigma'$ relation. We add error transitions to this as follows

$$\frac{\Sigma(L) = T \qquad T \neq T'}{[\Sigma]\,(L,T');\varphi \to \mathtt{error}}$$

We define the judgement $\vdash P : \Phi$ using a type and effect system in the usual way modulo the rule for $\mathtt{acc}^l$ expressions as discussed above. This judgement says that the system consists of well-typed threads

whose effects make up Φ. Our global check on the consistency of the local constraints in the effects is expressed in the rule:

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

where

$$\begin{array}{rcl} \mathsf{int}(\varphi_1 \parallel \ldots \parallel \varphi_n) & \stackrel{\text{def}}{=} & \bigcup_{i=1}^{n}\{(l,T);\varphi' | \varphi_i = (l,T);\varphi_i' \wedge \varphi' \in \mathsf{int}(\varphi_1 \parallel \ldots \parallel \varphi_i' \parallel \ldots \parallel \varphi_n,\})\\ \mathsf{compGen}(\varphi_1 \parallel \ldots \parallel \varphi_n,\Sigma) & \stackrel{\text{def}}{=} & \forall \varphi \in \mathsf{int}(\varphi_1 \parallel \ldots \parallel \varphi_n)\,.\,[\Sigma]\,\varphi \not\stackrel{*}{\to} [\_]\,\mathtt{error} \end{array}$$

In order to guarantee type safety we need to verify that each resource access returns a value of the expected type annotated on the effect of that resource access. This must be true irrespective of the interleaving of accesses which may occur before it. In the worst case we must look at all possible interleavings of a parallel effect and ensure that none of these reduce to an error under the resource reduction semantics. However, it may be that we can find a predicate over abstract states and effects that implies this property. We discuss this point further in the next section but for now we state our main properties of the generalised analysis:

Theorem 3.1. *(Subject Reduction and Fidelity) If* $\vdash [\sigma]P\colon \Phi$ *and* $[\sigma]\,P \to [\sigma']\,P$ *then there exists some* Φ' *such that* $\vdash [\sigma']P'\colon \Phi'$ *and* $[\Lambda(\sigma)]\,\Phi \to [\Lambda(\sigma')]\,\Phi'$

4 Invariability Proofs

The $\mathsf{compGen}(\Sigma,\Phi)$ predicate is a formal statement of the safety of possible interleavings of the effects on the resources Σ. Establishing $\mathscr{G}$ amounts to model checking the space of all traces on $[\Sigma]\,\Phi$. Whilst this approach has exponential complexity, it is a general solution where we have no additional knowledge about the structure of the resources, the permissible accesses, or the reduction semantics of the resources. Given more information about the resource's reduction relation then we may be able to define some other check which is much easier to establish but still implies $\mathscr{G}$ in that system. We illustrate this with the following examples.

Blocking Message Passing. In order to define a less costly check than model checking we consider a resource model of shared channels and channel queues with resource accesses $c!\langle v\rangle$ and $c?()$. It is easy to define an appropriate resource model and a corresponding abstract resource model for these: the states represent the current state of the channel queues and the actions move values in and out of these queues. The receive action has a blocking semantics so it has no action on an empty queue but will return the value at the head of the queue otherwise. The send action always returns a unit value. Suppose we know that each channel is shared between at most two threads and that threads use channels uni-directionally. Let us also suppose that systems always start with empty queues. Using this information we can define a *complementary* relation [4]:

Definition 4.1. *Complementary relation for blocking message passing systems:*

$$\begin{array}{rcl} \mathsf{compl}((c!\langle T\rangle,T');\varphi_1,(c?(),T'');\varphi_2) & \stackrel{\text{def}}{=} & T' = \mathsf{Unit} \wedge T = T'' \wedge \mathsf{compl}(\varphi_1,\varphi_2)\\ \mathsf{compl}(\varphi_1,\varphi_2) & = & \varphi_1 = \varepsilon \vee \varphi_2 = \varepsilon \end{array}$$

Given this we define compatibility as:

Definition 4.2. *Compatibility for blocking message passing systems:*

$$\mathsf{compatible}(\Sigma,\Phi) \stackrel{\text{def}}{=} \forall c.\mathsf{compl}(\varphi_1 \upharpoonright c, \varphi_2 \upharpoonright c) \vee \mathsf{compl}(\varphi_2 \upharpoonright c, \varphi_1 \upharpoonright c)$$

where $\varphi \upharpoonright c$ projects the effect φ to just the actions using channel c and φ_1 and φ_2 are the effects of the two threads that use c.

Given this definition it is not too hard to show that $\mathsf{compatible}(\Sigma_\varepsilon,\Phi)$ implies $\mathscr{G}(\Sigma_\varepsilon,\Phi)$ where Σ_ε is the abstract state representing empty channel queues. We can use this predicate in establishing that blocking message passing systems are well-typed, and hence are type safe, from the initial resource state. Subject reduction tells us that well-typed systems stay safe under reduction. This check is linear in the size of the effects which use the channel.

We sketch the proof that $\mathsf{compatible}(\Sigma_\varepsilon,\Phi)$ implies $\mathscr{G}(\Sigma_\varepsilon,\Phi)$ as follows. We know that only two threads perform accesses using that channel. In combination with our knowledge from the semantics that only actions using a channel can modify that channel's queue we can consider only the possible interleavings of the actions of these two threads which make use of a given channel ($\varphi_1 \upharpoonright c$ and $\varphi_2 \upharpoonright c$ respectively). If the usage is uni-directional, then irrespective of the interleavings the latter can only perform receive actions after the former performs a send action, as receive is a blocking action. In combination with FIFO access semantics then the receiver will receive values in the order that they are sent by the sender. This is true irrespective of whether all the sends are performed first and then all the the receives, whether the sends and receives interchange, or any other interleaving. Hence we preserve compatibility.

Non-Blocking Message Passing. A slightly different example is a message passing system where the receive actions are non-blocking. This uses the same resource and abstract models as blocking message passing, with one exception; the receive action has a non-blocking semantics where it returns a default (unit) value when performed on an empty queue and the head of the queue otherwise. We again suppose that two threads use a given channel uni-directionally and that we start with empty queues. Then the complementary behaviour to sending a value is polling a channel until a non-default value is received. In order to represent polling we need to add recursion and (external) choice to the language. We use the notation $\mu\underline{\mathsf{h}}.\varphi$ to define recursion where the recursive variable $\underline{\mathsf{h}}$ is bound in φ and the notation $\varphi_1 \& \varphi_2$ to define external choice, which reduces as follows:

$$\frac{\varphi_i \longrightarrow \varphi_i' \quad i \in 1,2}{\varphi_1 \& \varphi_2 \xrightarrow{(L,T)} \varphi_i'} \text{(:EExtChoice)}$$

Using this information we can define another complementary relation:

Definition 4.3. *Complementary relation for non-blocking message passing systems:*

$$\begin{array}{rcl}
\mathsf{compl}((c!\langle T\rangle, T'); \varphi_1, \mu\underline{h}.(c?(T''); \varphi_2 \& c?(\textsc{Unit}); \underline{h})) & \stackrel{\text{def}}{=} & T' = \mathsf{Unit} \wedge T = T'' \wedge \mathsf{compl}(\varphi_1,\varphi_2) \\
\mathsf{compl}(\varphi_1,\varphi_2) & = & \varphi_1 = \varepsilon \vee \varphi_2 = \varepsilon \\
\mathsf{compl}(\mu\underline{h}_1.\varphi_1, \mu\underline{h}_2.\varphi_2) & = & \mathsf{compl}(\varphi_1,\varphi_2) \\
\mathsf{compl}(\underline{h},\underline{h}) & = & \mathsf{true}
\end{array}$$

We can again show that $\mathsf{compatible}(\Sigma_\varepsilon,\Phi)$ implies $\mathscr{G}(\Sigma_\varepsilon,\Phi)$. As before we can only need to consider the effect of the two threads which are communicating uni-directionally using the channel. In order to guarantee that in the case of receiving a unit value the receiver doesn't go on to performing the next receive in its sequence we require it to perfom polling and wait to receive the non-default value. Hence

the receiver will receive all the values in the order sent. This may seem like a re-engineering of blocking receive, but recall that we are working with effects projected onto a specific channel; the receiver could go off and do some other behaviour on other channels before looping round and this would be invisible when performing the complementary check for this channel.

Bounded message passing To demonstrate the robustness of our technique we consider a resource model with a different semantics. In this case, the $c!\langle v\rangle$ action will block if the channel queue is full with K messages waiting in it. The semantics for this are easy to define. The changes we need to make to our analysis lie in the $\mathsf{compl}(\varphi_1, \varphi_2)$ predicate given above. In this case we simply introduce a counter in the definition clause for $\mathsf{compl}((c!\langle T\rangle, T'); \varphi_1, \varepsilon)$ so that the predicate becomes true when the counter reaches K. Again, it is easy to show that this predicate implies $\mathscr{G}$ and therefore may be used in establishing type-safety of such systems. In particular, we do not need to reprove subject reduction in this case.

5 Conclusion

In this paper we present a general resource model for resource accesses that return values of varying type. We provide a type and effect system for a multithreaded simply typed lambda calculus that features locally inferred return types as constraints in the effect annotations. These constraints are resolved globally between the multiple threads of the system in worst case by considering all possible interleavings of effects upon the resource state. This is tantamount to model checking and we provide examples in which we show that, given specific knowledge of the resource semantics, it is possible to define predicates that are easier to establish than the full model check yet nonetheless are sound. We believe that this approach clarifies the link between the semantics of the resources and the more routine aspects of the type safety proof and moreover allows reuse of the subject reduction and fidelity results for the general system.

The technical details of this work are included in our technical report [1] where we also include details of how to handle internal and external choice. In future work we intend to explore dynamic software updating for such systems and how the semantics of specific systems can inform safety proofs in the same way that the semantics informs compatibility definitions.

References

[1] Gabrielle Anderson. Behavioural Properties and Dynamic Software Updating- Thesis Report, http://eprints.ecs.soton.ac.uk/21995/, 2011.

[2] 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.

[3] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. *Session Types for Object-Oriented Languages*, book part (with own title) 20, pages 328–352. Lecture Notes in Computer Science. 2006.

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

[5] Kohei Honda, Makoto Kubo, and Vasco Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *In ESOP'98, volume 1381 of LNCS*, volume 171, pages 122–138, July 1998.

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

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

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