Semantics of Asynchronous C$^\sharp$

Anirudh Santhiar Aditya Kanade
Indian Institute of Science
{anirudh_s,kanade}@csa.iisc.ernet.in

This document accompanies the work on static deadlock detection for asynchronous C$^\sharp$ programs [2].

1. Semantics of Asynchronous C$^\sharp$

In this section, we first introduce the syntax of asynchronous operations in C$^\sharp$. Then, we discuss the semantics proposed by Bierman et al. [1] for asynchronous C$^\sharp$ programs. We explain how to extend their semantic domains and the runtime to capture the scheduling of continuations onto threads. Finally, we modify the rules defining the transition relation of the program to work with our augmented runtime. The modified rules, together with a new rule, serve to explicate scheduling aspects.

1.1 Asynchrony in C$^\sharp$

C$^\sharp$ has provided primitives to simplify asynchronous programming ever since the 5.0 release. Support for asynchrony requires two syntactic extensions to C$^\sharp$'s grammar shown below.

$$
\begin{array}{lcll}
pd & ::= & \texttt{async}\ \gamma\ m(\overline{\alpha\ x})\ \{\overline{\beta\ y}; \bar{s}\} & \text{[Procedure Decl.]} \\
e & ::= & (\texttt{await}_p \mid \texttt{await}_d)\ x & \text{[Await Expression]}
\end{array}
$$

First, procedures can be declared as asynchronous using the `async` modifier, as shown in the production for the non-terminal pd. Here, m refers to the procedure name and $\overline{\alpha\ x}$ to the sequence of parameter declarations, with α standing for the type declarations and x for the parameter names. This is followed by the procedure body, declarations of the locals $\overline{\beta\ y}$, and $\bar{s}$, the statements in the procedure. The return type γ = `Task<`δ`>` of an asynchronous procedure, is used to encapsulate the *future result* of an asynchronous procedure. Asynchronous procedures with the formal return type `Task<`δ`>` return objects of that type, even though they have a return statement returning a value of type δ. We will simply call the returned object as a *task*.

The `await` operation is the second syntactic extension to C$^\sharp$ to facilitate asynchronous programming. A procedure can asynchronously wait for the completion of a task (returned to it by another procedure) using the `await` operation; the variable x following the `await` must be of type `Task<`δ`>`[1]. The

[1] This is to simplify the presentation. The variable may be of any type implementing the *awaitable* interface [1].

$$
\begin{array}{rl}
\alpha, \beta, \gamma, \delta \in \tau = & \text{Finite set of types} \\
x, y, z \in Vars = & \text{Finite set of variables} \\
o \in Address = & \text{Infinite set of addresses} \\
v \in Value = & \text{Constant or Address} \\
t \in Tid = & \text{Set of thread identifiers} \\
l \in Label = & \text{Label } s \text{ (synchronous activation record)} \\
\mid & a(o) \text{ (asynchronous activation record)}
\end{array}
$$

Figure 1. Domains used by FCS5, augmented with the set Tid of thread identifiers.

`await` instructions may only be placed within asynchronous procedures, that is, procedures declared as `async`. The subscripts p and d on the `await` operator stand for "preserve" and "drop" respectively; these terms were introduced in Section 3.1 of the paper.

1.2 Preliminaries

Bierman et al. give an operational semantics for Asynchronous C$^\sharp$ dubbed Featherweight C$^\sharp$ 5.0 (FCS5) [1]. At a high level, FCS5 models an asynchronous program as a set of threads mutating a shared heap. Values in the program are constants or addresses. The heap maps addresses to objects. Every thread comes equipped with a stack to support synchronous procedure calls. Both synchronous procedure calls and continuations are modeled as *activation records* in the runtime. An activation record tracks the current assignments of local variables together with a sequence of pending program statements to be executed. We extend this model in the following ways.

- We introduce explicit thread identifiers to track threads as part of the semantic domains.
- We model the buffers used by threads for holding pending callbacks to be executed on the thread, as part of the runtime state.
- We tag continuations corresponding to `await` statements with an identifier to track the thread they will be scheduled on.

$$
\begin{array}{rcll}
\rho & = & Vars \to Value & \text{[Store]} \\
\langle \rho, \bar{s} \rangle_t^l \in AR & = & Store \times Stmt^* \times Label \times Tid & \\
 & & & \text{[Activation Record]} \\
\phi \in Object & = & \langle \delta, F \rangle & \text{[Class object]} \\
 & \mid & \langle \gamma, \mathsf{finished}, v \rangle & \text{[Task, complete]} \\
 & \mid & \langle \gamma, \mathsf{active}, \mathbb{P}(AR) \rangle & \text{[Task, active]} \\
h & = & Address \to Object & \text{[Heap]} \\
\mu & = & Tid \to AR^* & \text{[Thread to Stack]} \\
\xi & = & Tid \to \mathbb{P}(AR) & \text{[Thread to Buffer]} \\
\sigma \in \Sigma & = & \langle h, \mu, \xi \rangle & \text{[State]}
\end{array}
$$

Figure 2. Runtime of FCS5, together with our extensions.

Figure 1 shows the semantic domains used by FCS5, and 2 shows the runtime of FCS5, together with our extensions. We use α, β, γ and δ to range over types, and x, y and z to refer to variables. Let the set Tid refer to identifiers associated with a finite set of threads created by the runtime in order to run the application (e.g., the main thread and threads belonging to the thread-pool). We use S to denote stacks, and B to denote buffers. We use ϵ to denote empty stacks. We shall now discuss the runtime in detail.

Modeling Threads Every thread has a runtime stack for synchronous procedure calls, and a buffer to be used for storing pending callbacks. Both synchronous procedure calls, and pending callbacks are represented as activation records. An activation record $R = \langle \rho, \bar{s} \rangle_t^l$ contains the store ρ mapping locals to values, the sequence of statements $\bar{s}$, a label l and a thread identifier t for the thread executing this activation record. A label l, is either s to denote an activation record for a synchronous procedure call, or $a(o)$ for an asynchronous procedure whose associated `Task` is stored at address o. The buffer of pending callbacks associated with t is given by $\xi(t)$. The function $\mu(t)$ returns the stack S associated with a thread identified by t, holding activation records that represent called functions that have not returned yet. A stack with activation record R at the head, and tail S is denoted using $R \circ \mathsf{S}$. We do not model thread creation since our focus is on the asynchrony semantics. The model can be extended to account for dynamic thread creation in the standard manner.

Heap Model The shared heap h maps an address to an object. An object can refer to either a regular class object, or a `Task` object which has special modeling. A class object $\langle \delta, F \rangle$ denotes an object of type δ with a partial map F from its fields to values. A task object represents the future result of an asynchronous procedure. It has an associated state. It is either (i) $\langle \gamma, \mathsf{finished}, v \rangle$, meaning that the associated asynchronous procedure, whose result it represents, has completed with value v of type γ, or (ii) $\langle \gamma, \mathsf{active}, \mathsf{B} \rangle$ meaning that the associated asynchronous procedure is in flight, with B denoting a buffer containing callbacks to be run when it completes.

The life-cycle of a task heap object is as follows: on creation, it possesses the state $\langle \gamma, \mathsf{active}, \emptyset \rangle$, meaning that there is no callback to invoke once it completes. It can transition from state $\langle \gamma, \mathsf{active}, \mathsf{B} \rangle$ to $\langle \gamma, \mathsf{active}, \mathsf{B} \cup \{R\} \rangle$, adding callback R to the buffer of callbacks to be invoked when it completes. This transition may arise because of awaiting the task – the continuation following the `await` statement will then be added to the buffer of callbacks. It can also transition, on completion, from $\langle \gamma, \mathsf{active}, \mathsf{B} \rangle$ to $\langle \gamma, \mathsf{finished}, v \rangle$ as it removes the associated callbacks and schedules them; $\langle \gamma, \mathsf{finished}, v \rangle$ is the final state of the task, where v is a value of type γ. There are no transitions out of the final state. A task in the final state is said to be *complete*.

1.3 Operational Semantics

As observed earlier, an asynchronous C$^\sharp$ program is modeled as a set of threads mutating a shared heap. The state of the program tracks the heap, in addition to the current states of all the threads. The state associated with a thread includes the stack of activation records, and the buffer of pending callbacks.

DEFINITION 1 (Runtime State). *The runtime state σ of the program P is a triple $\langle h, \mu, \xi \rangle$, where h, μ and ξ are as defined in Figure 2.*

Let $t_m \in Tid$ denote the main thread. The program begins executing in an initial state σ_0 where the buffers associated with all the threads are empty, and the stacks associated with all the threads except t_m are empty. The stack $\mu(t_m)$ contains only the record $\langle h_0, \bar{s} \rangle_{t_m}^s$ corresponding to the `Main` procedure of the program.

DEFINITION 2 (Transition System). *Let $\mathcal{T} = (\Sigma, \to, \sigma_0)$ be the transition system representing an asynchronous C$^\sharp$ program. Here, Σ is the set of states, $\to \subseteq \Sigma \times Stmt \times \Sigma$ represents the transitions between states, and σ_0 is the initial state. The transitions are governed by the rules given in Figure 3.*

In the following, we restrict our attention to language constructs central to asynchrony, referring the reader to Bierman et al. [1] for the treatment of other standard constructs. We present a minimal extension to FCS5, called FCS5*, to track the thread on which continuations are to resume. The operational semantics for FCS5* is presented in Figure 3. The operations considered are (i) calls to and returns from asynchronous procedures, (ii) `await` statements, both when they admit synchronous control flow and when they suspend, (iii) potentially blocking access to the `Result` field of a task, and finally (iv) running of callbacks by a thread.

Notation We say $\sigma \to \sigma'$ if $(\sigma, s, \sigma') \in \to$, ie., the antecedent of some rule in Figure 3 holds for statement s and

$$\frac{t \in \mathit{Tid} \quad \mu(t) = \langle \rho, y_0 = y_1.m(\bar{z}); \bar{s}\rangle_t^l \circ \mathsf{S} \quad o \text{ fresh} \quad h(\rho(y_1)) = \langle \delta, F\rangle \quad \mathit{decl}(\delta, m) = \overline{\beta\ y}; \bar{u} : \overline{\alpha\ x} \to_a \gamma}{h = h[o \mapsto \langle \gamma, \mathsf{active}, \emptyset\rangle] \quad \rho' = [\bar{x} \mapsto \rho(\bar{z}), \bar{y} \mapsto \text{default}(\bar{\beta}), \mathtt{this} \mapsto \rho(y_1)] \quad \mu(t) = \langle \rho', \bar{u}\rangle_t^{a(o)} \circ \langle \rho[y_0 \mapsto o], \bar{s}\rangle_t^l \circ \mathsf{S}} \text{ [Async-Invoke]}$$

$$\frac{t \in \mathit{Tid} \quad \mu(t) = \langle \rho, x = \mathtt{await}_*\ y; \bar{s}\rangle_t^{a(o)} \circ \mathsf{S} \quad \rho(y) = o_1 \quad h(o_1) = \langle \gamma, \mathsf{active}, \mathsf{B}\rangle \quad t_r = \mathsf{Schedule}(t, \mathtt{await}_*)}{\mu(t) = \mathsf{S} \quad h = h[o_1 \mapsto \langle \gamma, \mathsf{active}, \mathsf{B} \cup \{\langle \rho, x = y.\mathtt{Result}; \bar{s}\rangle_{t_r}^{a(o)}\}\rangle]} \text{ [Await-Suspend]}$$

$$\frac{\begin{array}{c} t \in \mathit{Tid} \quad \mu(t) = \langle \rho, x = \mathtt{await}_*\ y; \bar{s}\rangle_t^{a(o)} \circ \mathsf{S} \\ h(\rho(y)) = \langle \gamma, \mathsf{finished}, v\rangle \end{array}}{\mu(t) = \langle \rho[x \mapsto v], \bar{s}\rangle_t^{a(o)} \circ \mathsf{S}} \text{ [Await-Continue]} \qquad \frac{\begin{array}{c} t \in \mathit{Tid} \quad \mu(t) = \langle \rho, x = y.\mathtt{Result}; \bar{s}\rangle_t^l \circ \mathsf{S} \\ h(\rho(y)) = \langle \gamma, \mathsf{finished}, v\rangle \end{array}}{\mu(t) = \langle \rho[x \mapsto v], \bar{s}\rangle_t^l \circ \mathsf{S}} \text{ [Result-Complete]}$$

$$\frac{\begin{array}{c} t \in \mathit{Tid} \quad \mu(t) = \langle \rho, x = y.\mathtt{Result}; \bar{s}\rangle_t^l \circ \mathsf{S} \\ h(\rho(y)) = \langle \gamma, \mathsf{active}, \mathsf{B}\rangle \end{array}}{\mu(t) = \langle \rho, x = y.\mathtt{Result}; \bar{s}\rangle_t^l \circ \mathsf{S}} \text{ [Result-Block]} \qquad \frac{t \in \mathit{Tid} \quad \mu(t) = \epsilon \quad \xi(t) = \{R\} \cup \mathsf{B}}{\mu(t) = R \quad \xi(t) = \mathsf{B}} \text{ [Run-Callback]}$$

$$\frac{t, t_1, \ldots, t_n \in \mathit{Tid} \quad \mu(t) = \langle \rho, \mathsf{return}\ y; \bar{s}\rangle_t^{a(o)} \circ \mathsf{S} \quad h(o) = \langle \gamma, \mathsf{active}, \{b_1, b_1, \ldots, b_n\}\rangle \text{ where } b_i = \langle \rho_i, \bar{s}_i\rangle_{t_i}^{a(o_i)}}{\mu(t) = \mathsf{S} \quad h = h[o \mapsto \langle \gamma, \mathsf{finished}, \rho(y)\rangle] \quad \xi(t_i) = \xi(t_i) \cup \{\langle \rho_i, \bar{s}_i\rangle_{t_i}^{a(o_i)}\}} \text{ [Async-Return]}$$

Figure 3. Semantics of asynchronous operations in FCS5*.

σ' is the result of updating σ as per the rule. We only refer to the parts of the state updated in the consequents of the rules; the remaining components are not altered. Multiple consequents in the rules mean that the rule updates many components of the state. We use $f[y \mapsto v]$ to denote $\lambda x.\ if\ x = y\ then\ v\ else\ f(x)$, and $\mathtt{await}_*$ to mean either $\mathtt{await}_p$ or $\mathtt{await}_d$.

Transition Rules We begin by considering calls to asynchronous procedures. The rule [Async-Invoke] shows the transition taken on encountering a statement $y_0 = y_1.m(\bar{z})$, where the rhs targets an asynchronous procedure. The procedure to be invoked is determined based on the receiver's runtime type. The auxiliary function decl retrieves the procedure body $\overline{\beta\ y}; \bar{u}$ and type signature $\overline{\alpha\ x} \to_a \gamma$, with $\to_a$ identifying the procedure as async. A fresh task object is created at address o to represent the result of this invocation. The task object is in the active state initially with no callbacks registered. A new activation record containing the body of the procedure is pushed onto the stack, with its store reflecting correct values for formal parameters (including `this`) and locals. The frame represents an asynchronous procedure. Therefore, it is tagged with the label $a(o)$. We modify the corresponding rule in FCS5 to also track the original thread identifier t, since execution continues on the calling thread. Finally, the caller's store is updated to track o.

Await statements could either suspend (if the awaited task is incomplete), or admit synchronous flow otherwise. Rule [Await-Suspend] considers the case of awaiting a task that is in flight. In this case, the asynchronous procedure executing the `await` instruction must suspend. Its activation record is popped from the stack and added to the buffer of callbacks associated with the task being awaited. We introduce the Schedule function that determines the thread t_r where the callback will be run, and is defined as

$$\mathsf{Schedule}(t, \mathtt{await}_*) = \begin{cases} t & \text{if } * \equiv p \\ t', t' \in \text{thread-pool}, & \text{if } * \equiv d \end{cases}$$

If the `await` statement was configured to *preserve* context, then we return the original thread t. Otherwise we return an arbitrary thread-pool thread. Our extended semantics also allows encoding other scheduling policies by defining the Schedule function appropriately. The Schedule function defined above encodes the most common case observed in our benchmarks. The first action of the callback on resumption is to retrieve the result of the awaited task. Rule [Await-Continue] accounts for the case of synchronous control flow; it simply reads the value from the task being awaited and moves to the next statement, since the task is already complete in this case.

To show examples of invoking potentially *blocking* procedure on tasks, we give the rules [Result-Complete] and [Result-Block] for the `Result` field-access on tasks. If the task has completed, the procedure simply returns the result. If the task is active, then `Result` is a blocking call, modeled as a transition to the same state.

When the stack of a thread is empty, it may pick a callback from its buffer and begin executing the callback according to rule [Run-Callback]. Prior to returning from an asynchronous procedure, we need to schedule all the callbacks associated with its task. In FCS5 continuations are always scheduled on the thread-pool. However, in rule [Async-Return] callbacks are scheduled by posting the callback to the buffer associated with the thread it was registered to run on earlier. The rule also stores the return value in the task and changes the state of the task associated with the returning procedure to "finished". In practice, this involves a call to a *signaling* procedure, but for ease of presentation, we do not model this detail. In our model, a transition according to the [Async-Return] rule is analogous to a signaling procedure. Finally, the semantics presented is an interleaving semantics; at each step, a thread may be selected non-deterministically, and an applicable transition applied.

2. Deadlocks

Deadlocks in FCS5* arise because of the interaction of blocking and non-blocking waits on tasks. We can characterize these deadlocks as a necessary condition on the runtime state as follows.

DEFINITION 3 (Deadlock). *A set of threads $D \subseteq Tid$ is said to deadlock when each thread in D is blocked and the operation it depends on for unblocking may only run in the future on a blocked thread in D.*

References

[1] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'N' Play: Formalizing asynchronous C#. ECOOP, pages 233-257. Springer-Verlag, 2012

[2] A. Santhiar, A. Kanade. Static deadlock detection for asynchronous C# programs. PLDI, 2017.