Compiling Cooperative Task Management to Continuations

Keiko Nakata, Andri Saar

Compiling cooperative task management to continuations

Keiko Nakata and Andri Saar

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn

Abstract. Although preemptive concurrency models are dominant for multi-threaded concurrency, they may be criticized for the complexity of reasoning because of the implicit context switches. The actor model and cooperative concurrency models have regained attention as they encapsulate the thread of control. In this paper, we formalize a continuation-based compilation of cooperative multitasking for a simple language and prove its correctness.

1 Introduction

In a preemptive concurrency model, threads may be suspended and activated at any time. While preemptive models are dominant for multi-threaded concurrency, they may be criticized for the complexity of reasoning because of the implicit context switches. The programmer often has to resort to low-level synchronization primitives, such as locks, to prevent unwanted context switches. Programs written in such a way tend to be error-prone and are not scalable. The actor model [2] addresses this issue. Actors encapsulate the thread of control and communicate with each other by sending messages. They are also able to call blocking operations such as sleep, await and receive, reminiscent of cooperative multi-tasking. Erlang and Scala actors support actor-based concurrency models.

Creol [8] and ABS [7] combine a message-passing concurrency model and a cooperative concurrency model. In Creol, each object encapsulates a thread of control and objects communicate with each other using asynchronous method calls. Asynchronous method calls, instead of messages-passing, provide a type-safe communication mechanism and are a good match for object-oriented languages [4,3]. ABS generalizes the concurrency model of Creol by introducing *concurrent object groups* [10] as the unit of concurrency. The concurrency model of ABS can be split in two: in one layer, we have local, synchronous and shared-memory communication[1] in one concurrent object group (COG) and on the second layer we have asynchronous message-based concurrency between different concurrent object groups as in Creol. The behavior of one COG is based on the cooperative multitasking of external method invocations and internal

[1] In ABS, different tasks originating from the same object may communicate with each other via fields of the object.

method activations, with concrete scheduling points where a different task may get scheduled. Between different COGs only asynchronous method calls may be used; different COGs have no shared object heap. The order of execution of asynchronous method calls is not specified. The result of an asynchronous call is a future; callers may decide at run-time when to synchronize with the reply from a call. Asynchronous calls may be seen as triggers that spawn new method activations (or tasks) within objects. Every object has a set of tasks that are to be executed (originating from method calls). Among these, at most one task of all the objects belonging to one COG is active; others are suspended and awaiting execution. The concurrency models of Creol and ABS are designed to be suitable in the distributed setting, where one COG executes on its own (virtual) processor in a single node and different COGs may be executed on different nodes in the network.

In this paper, we are interested in the compilation of cooperative multi-tasking into continuations, motivated to execute a cooperative multi-tasking model on the JVM platform, which employs a preemptive model. The basic idea of using continuations to manage the control behavior of the computation has been known from 80's [12,6], and is still considered as a viable technique [9,11,1]. This is particularly so, if the programming language supports first-class continuations, as in the case of Scala, and hence one can obviate manual stack management. The contribution of the paper is a correctness proof of such a compilation scheme. Namely, we create a simplified source language, by extending the While language with (synchronous) procedure calls and operations for cooperative multi-tasking (i.e., blocking operations and creation of new tasks) and define a compilation function from the source language into the target language, which extends While with continuation operations—the target language is sequential. We then prove that the compilation preserves the operational behavior from the source language to the target language.

The remainder of the paper is organized as follows. We define the source language and its operational semantics in the next section, and the target language and its operational semantics in Section 3. In Section 4, we present the compilation function from the source language to the target language and, in Section 5 we prove its correctness. We conclude in Section 6.

2 Source language

$$
\begin{aligned}
S ::= {} & x := e \mid u := e \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } e \texttt{ do } S \\
& \mid \texttt{ var } x = e \mid f(\overline{e}) \mid \texttt{await } e \mid \texttt{spawn } f\ \overline{e} \mid \texttt{return}
\end{aligned}
$$

Fig. 1. Syntax of the source language

In Figure 1, we define the syntax for the source language. We use the overline notation to denote sequences, with ϵ denoting an empty sequence. It is the

While language extended with (local) variable definitions, `var` $x = e$, procedure calls $f(\overline{e})$, the await statement `await` e, creation of a new task `spawn` $f\ \overline{e}$, and the return statement `return`. For simplicity, we syntactically distinguish local variable assignment $x := e$ and global variable assignment $u := e$. The statement `var` $x = e$ defines a (task) local variable x and initializes it with the value of e. The statement $f(\overline{e})$ invokes the procedure f with arguments $\overline{e}$, to be executed within the same task. The procedure does not return the result to the caller, but may store the result in a global variable. The statement `await` e suspends the execution of the current task, which can be resumed when the guard expression e evaluates to true. The statement `spawn` $f\ \overline{e}$ spawns a new task, which executes the body of f with arguments $\overline{e}$. (Hence, in contrast to procedure calls, which are synchronous, `spawn` $f\ \overline{e}$ is like an asynchronous procedure call executed in a new task.) The `return` statement is a runtime construct, not appearing in the source program, and will be explained later.

We assume disjoint supplies of local variables (ranged over by x), global variables (ranged over by u), and procedure names (ranged over by f). We assume a set of (pure) expressions, whose elements are ranged over by e. We assume the set of values to be the integers, non-zero integers counting as truth and zero as falsity. The metavariable v ranges over values. We have two kinds of states – local states and global states. A local state, ranged over by ρ, maps local variables to values; a global state, ranged over by σ, maps global variables to values. We denotes by $\emptyset$ an empty mapping, whose domain is empty. Communication between different tasks is achieved via global variables. For simplicity, we assume a fixed set of global variables. The notation $\rho[x \mapsto v]$ denotes the update of ρ with v at x, when x is in the domain of ρ. If x is not in the domain, it denotes a mapping extension. The notation $\sigma[u \mapsto v]$ denotes similar. We assume given an evaluation function $[\![e]\!]_{(\rho,\sigma)}$, which evaluates e in the local state ρ and the global state σ. We write $(\rho, \sigma) \models e$ and $(\rho, \sigma) \not\models e$ to denote that e is true, resp. false with respect to ρ and σ. A *stack* Π is a non-empty list of local states, whose elements are separated by semicolons. A stack grows leftward, i.e., the leftmost element is the topmost element.

A program P consists of a procedure environment env_F which maps procedure names to pairs of a formal argument list and a statement, and a global state which maps global variables to their initial values. The entry point of the program will be the procedure named main.

We define the operational semantics of the source language as a transition system on *configuration*s, in the style of structural operational semantics. A configuration *cfg* consists of an active task identifier n, a global variable mapping σ and a set of tasks Θ. A task has an identifier and may be in one of the three forms: a triple $\langle e, S, \Pi \rangle$, representing a task that is awaiting to be scheduled, where e is the guard expression, S the statement and Π its stack; or, a pair $\langle S, \Pi \rangle$, representing the currently active task; or, a singleton $\langle \Pi \rangle$, representing a terminated task.

$$\begin{array}{lll} \textit{Configuration } \textit{cfg} & ::= & n, \sigma \triangleright \Theta \\ \textit{Task sets} & \Theta & ::= n\langle e, S, \Pi \rangle \mid n\langle S, \Pi \rangle \mid n\langle \Pi \rangle \mid \Theta \parallel \Theta \end{array}$$

The order of tasks in the task set is irrelevant: the parallel operator $\|$ is commutative and associative. Formally, we assume the following structural equivalence:

$$\Theta \equiv \Theta \qquad \Theta \parallel \Theta' \equiv \Theta' \parallel \Theta \qquad \Theta \parallel (\Theta' \parallel \Theta'') \equiv (\Theta \parallel \Theta') \parallel \Theta''$$

Transition rules in the semantics are in the form $\mathrm{env}_F \vdash cfg \to cfg'$, shown in Figure 2. The first two rules (S-Cong and S-Equiv) deal with congruence and structural equivalence. The rules for assignment, `skip`, if-then-else and while are self-explanatory. For instance, in the rule S-Assign-Local, the task is of the form $n\langle S, \Pi' \rangle$ where $S = x := e$ and $\Pi' = \rho; \Pi$. Note that the topmost element of the stack Π is the current local state. The rules for sequential composition may deserve some explanation. If the first statement S_1 suspends guarded by e in the stack Π' with the residual statement S_1' to be run when resumed, then the entire statement $S_1; S_2$ suspends in $\langle e, S_1'; S_2, \Pi' \rangle$, where the residual statement now contains the second statement S_2 (S-Seq-Grd). If S_1 terminates in Π', then S_2 will run next in Π' (S-Seq-Fin). Otherwise, S_1 transfers to S_1' with the stack Π', so that $S_1; S_2$ transfers to $S_1'; S_2$ with the same stack (S-Seq-Step). The await statement immediately suspends (S-Await) the currently active task, enabling us to switch to some other task in accordance to the scheduling rules. An example of the await statement (and the scheduling rules) at work can be found in the example in Figure 3. The statement `spawn` $f\ \overline{e}$ creates a new task $n'\langle \texttt{true}, S, [\overline{x} \mapsto \overline{v}] \rangle$ with n' a fresh identifier (S-Spawn). The caller task continues to be active. The newly created task is suspended, guarded by `true`, and may get scheduled at scheduling points by the scheduling rules (see below). Procedure invocation $f(\overline{e})$ evaluates the arguments $\overline{e}$ in the current state, pushes into the stack the local state $[\overline{x} \mapsto \overline{v}]$, mapping the formal parameters to the actual arguments, and transfers to S;`return`, where S is the body of f (S-Call). The `return` statement pops the topmost element from the stack (S-Return). The local variable definition `var` $x = e$ extends the current local state with the newly defined variable and initializes it with the value of e (S-Var).

The last three rules deal with scheduling. If the current active task has terminated, then a new task whose guard evaluates to true is chosen to be active (S-Sched-Fin). When the active task suspends, a scheduling point is reached. The rule (S-Sched-Same) considers the case in which the same task is scheduled; the rule (S-Sched-Other) considers the case in which a different task is scheduled.

As an example, we will look at a program containing one global variable u with the initial value 0 and the following procedures:

$$\begin{aligned} \mathsf{f} &\mapsto u := 1 \\ \mathsf{main} &\mapsto u := 3; \texttt{spawn}\ \mathsf{f}\ \epsilon; \texttt{await}\ u = 1; u := 2 \end{aligned}$$

A detailed step, showing the full derivation, can be seen in Figure 4. A full execution trace, showing all intermediate configurations, is shown in Figure 3.

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta' \to n', \sigma' \triangleright \Theta''}{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta \parallel \Theta' \to n', \sigma' \triangleright \Theta \parallel \Theta''} \text{ S-Cong}$$

$$\frac{\Theta \equiv \Theta' \quad \mathrm{env}_F \vdash n, \sigma \triangleright \Theta' \to n', \sigma' \triangleright \Theta''' \quad \Theta'' \equiv \Theta'''}{\mathrm{env}_F \vdash n, \sigma \triangleright \Theta \to n', \sigma' \triangleright \Theta''} \text{ S-Equiv}$$

$$\frac{x \in \mathsf{dom}\,\rho}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle x := e, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho[x \mapsto [\![e]\!]_{(\rho,\sigma)}]; \Pi\rangle} \text{ S-Assign-Local}$$

$$\frac{u \in \mathsf{dom}\,\sigma}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle u := e, \Pi\rangle \to n, \sigma[u \mapsto [\![e]\!]_{(\rho,\sigma)}] \triangleright n\langle \Pi\rangle} \text{ S-Assign-Global}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \to n', \sigma' \triangleright n\langle e, S_1', \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \to n', \sigma' \triangleright n\langle e, S_1'; S_2, \Pi'\rangle \parallel \Theta} \text{ S-Seq-Grd}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \to n', \sigma' \triangleright n\langle \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \to n', \sigma' \triangleright n\langle S_2, \Pi'\rangle \parallel \Theta} \text{ S-Seq-Fin}$$

$$\frac{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1, \Pi\rangle \to n', \sigma' \triangleright n\langle S_1', \Pi'\rangle \parallel \Theta}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S_1; S_2, \Pi\rangle \to n', \sigma' \triangleright n\langle S_1'; S_2, \Pi'\rangle \parallel \Theta} \text{ S-Seq-Step}$$

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle S_1, \rho; \Pi\rangle} \text{ S-If-True}$$

$$\frac{(\rho, \sigma) \not\models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle S_2, \rho; \Pi\rangle} \text{ S-If-False}$$

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{while } e \texttt{ do } S, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle S; \texttt{while } e \texttt{ do } S, \rho; \Pi\rangle} \text{ S-While-True}$$

$$\frac{(\rho, \sigma) \not\models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{while } e \texttt{ do } S, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho; \Pi\rangle} \text{ S-While-False} \qquad \frac{}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{skip}, \Pi\rangle \to n, \sigma \triangleright n\langle \Pi\rangle} \text{ S-Skip}$$

$$\frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)} \quad n' \text{ is fresh}}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{spawn } f\ \overline{e}, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho; \Pi\rangle \parallel n'\langle \texttt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle} \text{ S-Spawn} \qquad \frac{}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{await } e, \Pi\rangle \to n, \sigma \triangleright n\langle e, \texttt{skip}, \Pi\rangle} \text{ S-Await}$$

$$\frac{x \notin \mathsf{dom}\,\rho}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{var } x = e, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho[x \mapsto [\![e]\!]_{(\rho,\sigma)}]; \Pi\rangle} \text{ S-Var} \qquad \frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)}}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle f(\overline{e}), \rho; \Pi\rangle \to n, \sigma \triangleright n\langle S; \texttt{return}, [\overline{x} \mapsto \overline{v}]; \rho; \Pi\rangle} \text{ S-Call}$$

$$\frac{}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \texttt{return}, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \Pi\rangle} \text{ S-Return} \qquad \frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi\rangle \parallel n'\langle e, S, \rho; \Pi'\rangle \to n', \sigma \triangleright n'\langle S, \rho; \Pi'\rangle \parallel n\langle \Pi\rangle} \text{ S-Sched-Fin}$$

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle S, \rho; \Pi\rangle} \text{ S-Sched-Same} \qquad \frac{(\rho', \sigma) \models e'}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \rho\rangle \parallel n'\langle e', S', \rho'\rangle \to n', \sigma \triangleright n'\langle S', \rho'\rangle \parallel n\langle e, S, \rho\rangle} \text{ S-Sched-Other}$$

Fig. 2. Semantics of the source language

.

$$
\begin{aligned}
&\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle u := 3; \texttt{spawn f } \epsilon; \texttt{await } u = 1; u := 2, \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{spawn f } \epsilon; \texttt{await } u = 1; u := 2, \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \\
&\quad \to \mathsf{f}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle u := 1, \emptyset\rangle \\
&\quad \to \mathsf{f}, [u \mapsto 1] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 1] \triangleright \mathsf{main}\langle \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 1] \triangleright \mathsf{main}\langle u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle \\
&\quad \to \mathsf{main}, [u \mapsto 2] \triangleright \mathsf{main}\langle \emptyset\rangle \parallel \mathsf{f}\langle \emptyset\rangle
\end{aligned}
$$

Fig. 3. The full execution trace of the example program

$$
\dfrac{\dfrac{\dfrac{}{\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1, \emptyset\rangle \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}, \emptyset\rangle}\ \text{S-Await}}{\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle}\ \text{S-Seq-Grd}}{\mathrm{env}_F \vdash \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle \texttt{await } u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle \to \mathsf{main}, [u \mapsto 3] \triangleright \mathsf{main}\langle u = 1, \texttt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \texttt{true}, u := 1, \emptyset\rangle}\ \text{S-Cong}
$$

Fig. 4. Example of a derivation in the source language

3 Target language

We proceed to the target language. In Figure 5, we present the syntax of the target language. Expressions of the target language contain, besides pure expressions, *continuations* $[S, \Pi]$, which are pairs of a statement S and a stack Π, and support for *guarded (multi)sets*: collections which contain pairs of an expression and a value. The expression stored with each element is called a *guard expression*, and is evaluated when we query the set: only elements whose guard expressions hold may be returned. There are five expressions in the language to work with guarded sets: an empty set $\emptyset$, checking whether a set is empty ($\texttt{isEmpty } e_s$), adding an element ($\texttt{add } e_s\ e_g\ e$), fetching an element ($\texttt{get } e_s$) and removing an element ($\texttt{del } e_s\ e$).

Similar to the source language, for the target language we extend While with local variable definitions and procedure calls. We also add delimited control operators, $\texttt{shift } k\ \{S\}$, $\texttt{reset } \{S\}$, $\texttt{invoke } k$ [5]. The statement $\texttt{shift } k\ \{S\}$ captures the rest of the computation, or continuation, up to the closest surrounding $\texttt{reset } \{\}$, binds it to k, and proceeds to execute S. In $\texttt{shift } k\ \{S\}$, k is a binding occurrence, whose scope is S. Hence, the statement $\texttt{reset } \{S\}$ delimits the captured continuation. The statement $\texttt{invoke } k$ invokes, or jumps into, the continuation bound to the variable k. The statement $\texttt{R } \{S\}$, where $\texttt{R}$ is a new constant, is a runtime construct to be explained later.

The target language is sequential and, unlike the source language, it contains no explicit support for parallelism. Instead, we provide building blocks – contin-

$$
\begin{array}{lll}
\textit{Expressions} & e & ::= \ldots \\
& & \mid\ [S, \Pi] \mid \emptyset \mid \texttt{isEmpty}\ e \mid \texttt{add}\ e_1\ e_2\ e_3 \mid \texttt{del}\ e_1\ e_2 \mid \texttt{get}\ e \\
\textit{Statements} & S & ::= x := e \mid u := e \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \\
& & \mid\ \texttt{var}\ x = e \mid f(\overline{e}) \mid \texttt{reset}\ \{S\} \mid \texttt{shift}\ k\ \{S\} \mid \texttt{invoke}\ k \\
& & \mid\ \texttt{return} \mid \texttt{R}\ \{S\}
\end{array}
$$

Fig. 5. Syntax of the target language

uations and guarded sets – that are used to switch between tasks and implement an explicit scheduler in Section 4.

We assume given an evaluation function $[\![e]\!]_{(\Pi,\sigma)}$ for pure expressions, which evaluates e with respect to the stack Π (the evaluation only looks at the current local state, which is the topmost element of Π) and the global state σ. In Figure 6, the evaluation function is extended to operations for guarded sets.

We define an operational semantics for the target language as a reduction semantics over configurations, using evaluation contexts. A configuration $\langle S, \Pi, \sigma \rangle$ is a triple of a statement S, a stack Π and a global state σ.

Evaluation contexts E are statements with a hole, specifying where the next reduction may occur. They are defined by

$$E ::= [] \mid E; S \mid \texttt{R}\ \{E\}$$

We denote by $E[S]$ the statement obtained by placing S in the hole of E.

We define basic reduction rules in Figure 6. `reset` $\{S\}$ inserts a marker † into the stack, just below the current state, and reduces to `R` $\{S\}$ to continue the execution of S (T-Reset). We use a marker to delimit the portion of the stack captured by `shift` and to align the stack when exiting from `R` $\{\}$. The runtime construct `R` $\{\}$ is used to record that the marker has been set. `shift` k $\{S\}$ captures the rest of the execution up to and including the closest surrounding `R` $\{\}$ together with the corresponding portion of the stack, binds it to a fresh variable k' in the local state, and continues with the statement S' obtained by substituting k by k' in S (T-Shift). The surrounding `R` $\{\}$ is kept intact. F is an evaluation context that does not intersect `R` $\{\}$, formally,

$$F ::= [] \mid F; S$$

Note that `shift` k $\{S\}$ captures the stack up to and including the topmost †, which has been inserted by the closest surrounding reset. Once the body of `R` $\{\}$ terminates, i.e., reduces to `skip`, then we remove the `R` $\{\}$ and pop the stack until the topmost †, but leaving the state just above † in the stack (T-R). `invoke` k invokes the continuation bound to k (T-Invoke). Namely, if k is bound to $[S, \Pi']$ in the local or global state, then the statement reduces to S;`return` and the stack Π' is pushed into the current stack. S must be necessarily of the form `R` $\{S'\}$, where S' does not contain `R`, and Π' contains exactly one † at the bottom. When exiting from the `R` $\{\}$, the state immediately above † in Π' will

$$[\![\texttt{add}\ e_s\ e_g\ e]\!]_{(\Pi,\sigma)} = [\![e_s]\!]_{(\Pi,\sigma)} \cup (e_g, [\![e]\!]_{(\Pi,\sigma)})$$

$$[\![\texttt{get}\ e_s]\!]_{(\Pi,\sigma)} = v \text{ when } \exists e_g \exists v.\ (e_g, v) \in [\![e_s]\!]_{(\Pi,\sigma)} \wedge [\![e_g]\!]_{(\Pi,\sigma)} = \texttt{true}$$

$$[\![\texttt{del}\ e_s\ e]\!]_{(\Pi,\sigma)} = [\![e_s]\!]_{(\Pi,\sigma)} \backslash (e_g, [\![e]\!]_{(\Pi,\sigma)}) \text{ when } \exists e_g.\ (e_g, [\![e]\!]_{(\Pi,\sigma)}) \in [\![e_s]\!]_{(\Pi,\sigma)}$$

$$[\![\texttt{isEmpty}\ e_s]\!]_{(\Pi,\sigma)} = \begin{cases} \texttt{true} & \text{if } [\![e_s]\!]_{(\Pi,\sigma)} = \emptyset \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\frac{}{\text{env}_F \vdash \langle \texttt{reset}\ \{S\}, \rho; \Pi, \sigma\rangle \to \langle \texttt{R}\ \{S\}, \rho\dagger; \Pi, \sigma\rangle}\ \text{T-Reset}$$

$$\frac{\Pi \text{ does not contain } \dagger \quad k' \text{ fresh} \quad S' \text{ is obtained from S by replacing } k \text{ with } k'}{\begin{array}{c}\text{env}_F \vdash \langle \texttt{R}\ \{F[\texttt{shift}\ k\ \{S\}]\}, \rho; \Pi\dagger; \Pi', \sigma\rangle \\ \to \langle \texttt{R}\ \{S'\}, \rho[k' \mapsto [\texttt{R}\ \{F[\texttt{skip}]\}, \rho; \Pi\dagger]]; \Pi\dagger; \Pi', \sigma\rangle\end{array}}\ \text{T-Shift}$$

$$\frac{\Pi \text{ does not contain } \dagger}{\text{env}_F \vdash \langle \texttt{R}\ \{\texttt{skip}\}, \Pi; \rho\dagger; \Pi', \sigma\rangle \mapsto \langle \texttt{skip}, \rho; \Pi', \sigma\rangle}\ \text{T-R}$$

$$\frac{[\![k]\!]_{(\Pi,\sigma)} = [S, \Pi']}{\text{env}_F \vdash \langle \texttt{invoke}\ k, \Pi, \sigma\rangle \to \langle S; \texttt{return}, \Pi'; \Pi, \sigma\rangle}\ \text{T-Invoke}$$

$$\frac{}{\text{env}_F \vdash \langle \texttt{skip}; S, \Pi, \sigma\rangle \to \langle S, \Pi, \sigma\rangle}\ \text{T-Skip}$$

$$\frac{\text{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\Pi,\sigma)}}{\text{env}_F \vdash \langle f(\overline{e}), \Pi, \sigma\rangle \to \langle S; \texttt{return}, [\overline{x} \mapsto \overline{v}]; \Pi, \sigma\rangle}\ \text{T-Call}$$

$$\frac{}{\text{env}_F \vdash \langle \texttt{return}, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma\rangle}\ \text{T-Return}$$

$$\frac{(\Pi, \sigma) \models e}{\text{env}_F \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \Pi, \sigma\rangle \to \langle S_1, \Pi, \sigma\rangle}\ \text{T-If-True}$$

$$\frac{(\Pi, \sigma) \not\models e}{\text{env}_F \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \Pi, \sigma\rangle \to \langle S_2, \Pi, \sigma\rangle}\ \text{T-If-False}$$

$$\frac{(\Pi, \sigma) \models e}{\text{env}_F \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle \to \langle S; \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle}\ \text{T-While-True}$$

$$\frac{(\Pi, \sigma) \not\models e}{\text{env}_F \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma\rangle}\ \text{T-While-False}$$

$$\frac{x \notin \mathsf{dom}\,\rho}{\text{env}_F \vdash \langle \texttt{var}\ x = e, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \rho[x \mapsto [\![e]\!]_{(\rho;\Pi,\sigma)}]; \Pi, \sigma\rangle}\ \text{T-Var}$$

$$\frac{x \in \mathsf{dom}\,\rho}{\text{env}_F \vdash \langle x := e, \rho; \Pi, \sigma\rangle \to \langle \texttt{skip}, \rho[x \mapsto [\![e]\!]_{(\rho;\Pi,\sigma)}]; \Pi, \sigma\rangle}\ \text{T-Assign-Local}$$

$$\frac{u \in \mathsf{dom}\,\sigma}{\text{env}_F \vdash \langle u := e, \Pi, \sigma\rangle \to \langle \texttt{skip}, \Pi, \sigma[u \mapsto [\![e]\!]_{(\Pi,\sigma)}]\rangle}\ \text{T-Assign-Global}$$

Fig. 6. Semantics of the target language

$$
\begin{aligned}
&\mathrm{env}_F \vdash \langle \texttt{reset}\ \{u' := 1; \texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset, [u \mapsto 0; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u' := 1; \texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset\dagger, [u \mapsto 0; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{shift}\ k\ \{u := k\}; u' := 0\}; \texttt{invoke}\ u, \emptyset\dagger, [u \mapsto 0; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u := k'\}; \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]]\dagger, [u \mapsto 0; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}\}; \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]]\dagger, [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{invoke}\ u, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}; u' := 0\}; \texttt{return}, \emptyset\dagger; \\
&\qquad [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{u' := 0\}; \texttt{return}, \emptyset\dagger; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 1]\rangle \\
&\quad \mapsto \langle \texttt{R}\ \{\texttt{skip}\}; \texttt{return}, \emptyset\dagger; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{return}, \emptyset; [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle \\
&\quad \mapsto \langle \texttt{skip}, [k' \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]], [u \mapsto [\texttt{R}\ \{\texttt{skip}; u' := 0\}, \emptyset\dagger]; u' \mapsto 0]\rangle
\end{aligned}
$$

Fig. 7. Capturing and invoking a continuation

be left in the stack, which is popped by the trailing `return`. An example of how to capture and invoke a continuation is shown in Figure 7. In the example, we assume that the variables u and u' are global.

Procedure call $f(\overline{e})$ reduces to S; `return` where S is the body of the procedure f, and pushes a local state $[\overline{x} \mapsto \overline{v}]$, binding procedure's formal arguments to actual arguments, into the stack (T-Call). The trailing `return` ensures that, once the execution of S; `return` terminates, the stack is aligned to the original Π. `return` pops the topmost element from the stack (T-Return). The remaining rules are self-explanatory.

Given the basic reduction rules, we now define a standard reduction, denoted by $\mapsto$, by

$$\frac{\mathrm{env}_F \vdash \langle S, \Pi, \sigma\rangle \to \langle S', \Pi', \sigma'\rangle}{\mathrm{env}_F \vdash \langle E[S], \Pi, \sigma\rangle \mapsto \langle E[S'], \Pi', \sigma'\rangle}$$

stating that the configuration $\langle S, \Pi, \sigma\rangle$ standard reduces to $\langle S', \Pi', \sigma'\rangle$ if there exist an evaluation context E and statement S_0 and S_0' such that $S = E[S_0]$ and $S' = E[S_0']$ and $\mathrm{env}_F \vdash \langle S_0, \Pi, \sigma\rangle \to \langle S_0', \Pi', \sigma'\rangle$. The standard reduction is deterministic.

4 Compilation

When compiling a program P into the target language, we compile expressions and statements according to the scheme shown in Figure 8. Expressions are translated into the target language as-is, and statements that have a corresponding equivalent in the target language are also translated in a straightforward manner. The two statements that have no direct correspondence in the target language are `await` and `spawn`. We look at how these statements are translated and how they interact with the scheduler later.

$$
\begin{aligned}
[\![e]\!] &= e \\
[\![x := e]\!] &= x := [\![e]\!] \\
[\![u := e]\!] &= u := [\![e]\!] \\
[\![\texttt{skip}]\!] &= \texttt{skip} \\
[\![S_1; S_2]\!] &= [\![S_1]\!]; [\![S_2]\!] \\
[\![\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2]\!] &= \texttt{if } [\![e]\!] \texttt{ then } [\![S_1]\!] \texttt{ else } [\![S_2]\!] \\
[\![\texttt{while } e \texttt{ do } S]\!] &= \texttt{while } [\![e]\!] \texttt{ do } [\![S]\!] \\
[\![\texttt{var } x = e]\!] &= \texttt{var } x = [\![e]\!] \\
[\![\texttt{await } e]\!] &= \texttt{shift } k \ \{T := \texttt{add } T \ [\![e]\!] \ k\}; \texttt{skip} \\
[\![f(\overline{e})]\!] &= f_S(\overline{[\![e]\!]}) \\
[\![\texttt{spawn } f \ \overline{e}]\!] &= f_A(\overline{[\![e]\!]})
\end{aligned}
$$

Fig. 8. Compilation of source programs

$$
\mathit{Sched} = \texttt{while } (\neg\texttt{isEmpty } T) \texttt{ do reset } \{k := \texttt{get } T; T := \texttt{del } T \ k; \texttt{invoke } k\}
$$

Fig. 9. Scheduler

The central idea of the compilation scheme is to use continuations to handle the suspension of tasks, and have an explicit *scheduler* (for brevity, in the examples we use *Sched* to denote the scheduler), shown in Figure 9. The control will pass to the scheduler every time a task either suspends or finishes, and the scheduler will pick up a new task to execute. During runtime, we also use a global variable T, which we assume not to be used by the program to be compiled. The global variable T stores the *task set*, corresponding to Θ in the source semantics, that contains all the tasks in the system. The tasks are stored as continuations with guard expressions.

The scheduler loops until the task set is empty (all tasks have terminated), in each iteration picking a continuation from T where the guard expression evaluates to `true`, removing it from T and then invoking the continuation using `invoke` k. The body of the scheduler is wrapped in a `reset`, guaranteeing that when a task suspends, the capture will be limited to the end of the current task. After the execution is completed – either by suspension or by just finishing the work – the control comes back to the scheduler.

Suspension (`await` e in the source language) is compiled to a `shift` statement. When evaluating the statement, the original computation until the end of the enclosing `R` {} will be captured and stored in the continuation k, and the original program is replaced with the body of the `shift`. The enclosing `R` {} guarantees that we capture only the statements up until the end of the current task, thus providing a facility to proceed with the execution of the task later.

$$
\begin{aligned}
[\![n, \sigma \triangleright n\langle \Pi \rangle \parallel \Theta]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2] \rangle \\
[\![n, \sigma \triangleright n\langle e, S, \Pi \rangle \parallel \Theta]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup [\![n\langle e, S, \rho \rangle]\!]_2] \rangle \\
[\![n, \sigma \triangleright n\langle S, \Pi \rangle \parallel \Theta]\!] &= \langle \texttt{R}\ \{\texttt{R}\ \{[\![S]\!]\}; \texttt{return}\}; \mathit{Sched}, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2] \rangle \\
[\![\Theta \parallel \Theta']\!]_2 &= [\![\Theta]\!]_2 \cup [\![\Theta']\!]_2 \\
[\![n\langle e, S, \Pi \rangle]\!]_2 &= \{([\![e]\!], [\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}, \Pi\dagger])\} \\
[\![n\langle \Pi \rangle]\!]_2 &= \emptyset
\end{aligned}
$$

Fig. 10. Compilation of configurations

The body of the `shift` statement simply takes the captured continuation, k, and adds it to the global task set, with the appropriate guard expression. After adding the continuation to the task set, the control passes back to the scheduler.

Procedures in env_F will get translated into two different procedures for synchronous and asynchronous calls, as follows:

$$
\begin{aligned}
f_S &\mapsto [\![S]\!] \\
f_A &\mapsto \texttt{reset}\ \{\texttt{shift}\ k\ \{T := \texttt{add}\ T\ \texttt{true}\ k\}; [\![S]\!]\}
\end{aligned}
$$

When making an asynchronous call, the body of the procedure will be immediately captured in a continuation, added to the global task set, and the control passes back to the invoker via the usual synchronous call mechanism.

The entry point of a program in the source language, main, is a regular procedure and will get translated according to the usual rules into two procedures, main_A and main_S. In the target language, we must invoke the scheduler, and thus we use a different entry point:

$$T := \emptyset; \mathsf{main}_A(); \mathit{Sched}$$

After initializing the task set to be empty, the first statement will add an asynchronous call to the original entry point of the program, and passes control to the scheduler. As there is only one task in the task set – the task that will invoke the original entry point – the scheduler will immediately proceed with that.

5 Correctness

In this section, we prove that our compilation scheme is correct in the sense that it preserves the operational behavior from the source program into the (compiled) target program. Specifically, we prove that reductions in the source language are simulated by corresponding reductions in the target language. To do so, we extend the compilation scheme to configurations in Figure 10.

The compilation scheme for configurations follows the idea of the compilation scheme detailed in Section 4. We have two compilation functions: $[\![\cdot]\!]_2$, which

$$\frac{\rho\ x = [S, \Pi']}{\mathrm{env}_F \vdash \langle \texttt{invoke}\ x, \rho; \Pi, \sigma\rangle \to \langle S; \texttt{return}, \Pi'; \rho\backslash x; \Pi, \sigma\rangle}\ \text{T-InvokeOnce}$$

Fig. 11. Alternative rule for `invoke`

generates a task set from Θ, and $[\![\cdot]\!]$, which generates a configuration in the target semantics.

Every suspended task in the task set Θ is compiled to a pair consisting of the compiled guard expression and a continuation that has been constructed from the original statement and stack. The statement is wrapped in a `R` {} block and we prepend a `skip` statement, just as it would happen when a continuation is captured in the target language.

If the active task is finished or is suspended (but no new task has been scheduled yet), the generated configuration will immediately contain the scheduler. If the task has suspended, the task is compiled according to the previously described scheme and appended to T. Active tasks are wrapped in two `R` {} blocks and the stack Π is concatenated on top of the local state of the scheduler.

When the scheduler invokes a continuation k, the continuation will stay in the local state of the scheduler until control comes back to the scheduler. This is unnecessary, as the value is never used after it has been invoked; furthermore, the variable is immediately assigned a new value after control passes back to the scheduler. Thus, as an optimization, we may switch to an alternative reduction rule for `invoke` k, which only allows a continuation to be used once, T-InvokeOnce, shown in Figure 11. Although the behavior of the program is equivalent under both versions, using the one-shot version also allows us to state the correctness theorem in a more concise and straightforward manner, as the local state of the scheduler will always be empty when we are currently executing some task. In the proof, we assume this rule to be used instead of the original T-Invoke rule.

The following lemma states that the compilation of statements is compositional with respect to evaluation contexts, where evaluation contexts for the source language are defined inductively by

$$K := [] \mid K; S$$

Lemma 1.

$$[\![K[S]]\!] = [\![K]\!][[\![S]\!]]$$

Proof. By induction on the structure of K. □

The correctness theorem below states that a one-step reduction in the source language is simulated by multiple-step reductions in the target language.

As an example, in Figure 12 we show the compiled form for both the initial and final configurations shown for the step in Figure 4 and in Figure 13, we show how to reach the compiled equivalent of the configuration in multiple steps in the target semantics.

$$
\begin{aligned}
&[\![\mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle \mathtt{await}\ u = 1; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \mathtt{true}, u := 1, \emptyset\rangle]\!] \\
&\quad = \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ (u = 1)\ k\}; \mathtt{skip}; u := 2\}; \mathtt{return}\}; \mathit{Sched}, \emptyset\dagger; \emptyset\dagger, \\
&\qquad\quad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&[\![\mathsf{main}, [u \mapsto 0] \triangleright \mathsf{main}\langle u = 1, \mathtt{skip}; u := 2, \emptyset\rangle \parallel \mathsf{f}\langle \mathtt{true}, u := 1, \emptyset\rangle]\!] \\
&\quad = \langle \mathit{Sched}, \emptyset, [u \mapsto 0; T \mapsto \{(u = 1, [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger]), (\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle
\end{aligned}
$$

Fig. 12. Example of compiling a configuration

$$
\begin{aligned}
&\langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ (u = 1)\ k\}; \mathtt{skip}; u := 2\}; \mathtt{return}\}; \mathit{Sched}, \emptyset\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{T := \mathtt{add}\ T\ (u = 1)\ k'\}; \mathtt{return}\}; \mathit{Sched}, [k' \mapsto [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger]]\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}\}; \mathtt{return}\}; \mathit{Sched}, [k' \mapsto [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger]]\dagger; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{return}\}; \mathit{Sched}, [k' \mapsto [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger]]; \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{skip}\}; \mathit{Sched}, \emptyset\dagger, \\
&\qquad [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle \\
&\mapsto \langle \mathit{Sched}, \emptyset, [u \mapsto 0; T \mapsto \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; u := 1\}, \emptyset\dagger]), (u = 1, [\mathtt{R}\ \{\mathtt{skip}; \mathtt{skip}; u := 2\}, \emptyset\dagger])\}]\rangle
\end{aligned}
$$

Fig. 13. Reduction of the compiled configuration

Theorem 1. *For all configurations cfg_S and cfg'_S such that*

$$\mathrm{env}_F \vdash cfg_S \to cfg'_S$$

holds, then the following must also hold:

$$[\![\mathrm{env}_F]\!] \vdash [\![cfg_S]\!] \mapsto^+ [\![cfg'_S]\!]$$

Proof. By induction over the derivation, analyzing the step taken. The possible steps have one of the following forms:

– Case

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \to n', \sigma' \triangleright n\langle \Pi'\rangle \parallel \Theta'$$

Rules matching this pattern are S-Assign-Local, S-Assign-Global, S-While-False, S-Spawn, S-Return, S-Var. As a representative example, we will look at S-Spawn in detail.

$$\frac{\mathrm{env}_F\ f = (\overline{x}, S) \quad \overline{v} = [\![\overline{e}]\!]_{(\rho,\sigma)} \quad n' \text{ is fresh}}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \mathtt{spawn}\ f\ \overline{e}, \rho; \Pi\rangle \to n, \sigma \triangleright n\langle \rho; \Pi\rangle \parallel n'\langle \mathtt{true}, S, [\overline{x} \mapsto \overline{v}]\rangle}$$

In this case, the source and target configurations are compiled to:

$$
\begin{aligned}
[\![cfg_S]\!] &= \langle \mathtt{R}\ \{\mathtt{R}\ \{f_A(\overline{e})\}; \mathtt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
[\![cfg'_S]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle
\end{aligned}
$$

Let the bottommost element of Π be ρ', where $\rho = \rho'$ if Π is empty. The compiled source configuration will reduce as follows:

$$
\begin{aligned}
&[\![\mathrm{env}_F]\!] \vdash \langle \mathtt{R}\ \{\mathtt{R}\ \{f_A(\overline{e})\}; \mathtt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{reset}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; [\![S]\!]\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}]; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; [\![S]\!]\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{T := \mathtt{add}\ T\ \mathtt{true}\ k\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, \\
&\qquad [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}\}; \mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]\dagger; \rho; \Pi\dagger; \emptyset\dagger, \\
&\qquad \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{return}\}; \mathtt{return}\}; \mathit{Sched}, [\overline{x} \mapsto \overline{v}, k \mapsto [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger]; \rho; \Pi\dagger; \emptyset\dagger, \\
&\qquad \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{R}\ \{\mathtt{skip}\}; \mathtt{return}\}; \mathit{Sched}, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{return}\}; \mathit{Sched}, \rho'; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathtt{R}\ \{\mathtt{skip}\}; \mathit{Sched}, \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle \\
&\mapsto \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{(\mathtt{true}, [\mathtt{R}\ \{\mathtt{skip}; [\![S]\!]\}, [\overline{x} \mapsto \overline{v}]\dagger])\}]\rangle
\end{aligned}
$$

The configuration we obtain from evaluation is exactly equal to the compiled configuration, thus for this case our claim holds.

– Case

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \to n', \sigma' \triangleright n\langle e, S', \Pi'\rangle \parallel \Theta'$$

There are only two possible rules: S-Seq-Grd and S-Await. In both cases, it must be that $\sigma = \sigma'$, $\Pi = \Pi'$, $\Theta \equiv \Theta'$ and there exists some K such that $S = K[\mathtt{await}\ e]$ and $S' = K[\mathtt{skip}]$. Therefore, taking into account Lemma 1, the source and target configurations are compiled to:

$$
\begin{aligned}
[\![cfg_S]\!] &= \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![K]\!][[\![\mathtt{await}\ e]\!]]\}; \mathtt{return}\}; \mathit{Sched}, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
&= \langle \mathtt{R}\ \{\mathtt{R}\ \{[\![K]\!][\mathtt{shift}\ k\ \{T := \mathtt{add}\ T\ [\![e]\!]\ k\}; \mathtt{skip}]\}; \mathtt{return}\}; \mathit{Sched}, \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto [\![\Theta]\!]_2]\rangle \\
[\![cfg'_S]\!] &= \langle \mathit{Sched}, \emptyset, \sigma[T \mapsto [\![\Theta]\!]_2 \cup \{([\![e]\!], \mathtt{R}\ \{[\![K]\!][\mathtt{skip}]\}, \Pi\dagger)\}\rangle
\end{aligned}
$$

An example of this reduction can be seen in Figure 13.

– Case

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle S, \Pi\rangle \parallel \Theta \to n', \sigma' \triangleright n\langle S', \Pi'\rangle \parallel \Theta'$$

Rules matching this pattern are S-Seq-Fin, S-Seq-Step, S-If-True, S-If-False, S-While-True, S-Call. In the case of S-Seq-Step, we know that $S = S_0; S_1$ and $S' = S'_0; S_1$. By induction hypothesis, we get that

$$[\![\mathrm{env}_F]\!] \vdash [\![n, \sigma \triangleright n\langle S_0, \Pi\rangle \parallel \Theta]\!] \to [\![n', \sigma' \triangleright n\langle S'_0, \Pi'\rangle \parallel \Theta']\!]$$

As by the definition of the compilation function $[\![S]\!] = [\![S_0]\!]; [\![S_1]\!]$ and $[\![S']\!] = [\![S'_0]\!]; [\![S_1]\!]$, we obtain the needed result:

$$[\![\mathrm{env}_F]\!] \vdash [\![n, \sigma \triangleright n\langle S_0; S_1, \Pi\rangle \parallel \Theta]\!] \to [\![n', \sigma' \triangleright n\langle S'_0; S_1, \Pi'\rangle \parallel \Theta']\!]$$

For S-Seq-Fin, we know that $S = S_0; S_1$ and $S' = S_1$. Then the case follows by analyzing the step taken to reduce S_0.
The other cases are straightforward.

– One of the following three:

$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle e, S, \Pi \rangle \parallel \Theta \to n', \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle S, \Pi \rangle \parallel \Theta$$
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e', S', \Pi' \rangle \parallel n'\langle e, S, \Pi \rangle \parallel \Theta \to n', \sigma \triangleright n\langle e', S', \Pi' \rangle \parallel n'\langle S, \Pi \rangle \parallel \Theta$$
$$\mathrm{env}_F \vdash n, \sigma \triangleright n\langle e, S, \Pi \rangle \parallel \Theta \to n, \sigma \triangleright n\langle S, \Pi \rangle \parallel \Theta$$

These three patterns match each of the scheduling rules. We will look only at the first one.

$$\frac{(\rho, \sigma) \models e}{\mathrm{env}_F \vdash n, \sigma \triangleright n\langle \Pi' \rangle \parallel n'\langle e, S, \rho; \Pi \rangle \to n', \sigma \triangleright n'\langle S, \rho; \Pi \rangle \parallel n\langle \Pi' \rangle}\ \text{S-Sched-Fin}$$

$$[\![cfg_S]\!] = \langle Sched, \emptyset, \sigma[T \mapsto \{([\![e]\!], [\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle$$
$$[\![cfg'_S]\!] = \langle \texttt{R}\ \{\texttt{R}\ \{[\![S]\!]\}; \texttt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle$$

The initial configuration will reduce as (with some of the steps omitted):

$[\![\mathrm{env}_F]\!] \vdash \langle \texttt{while}\ (\neg\texttt{isEmpty}\ T)\ \texttt{do reset}\ \{k := \texttt{get}\ T; T := \texttt{del}\ T\ k; \texttt{invoke}\ k\},$
$\quad \emptyset, \sigma[T \mapsto \{([\![e]\!], [\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle$
$\mapsto \langle \texttt{reset}\ \{k := \texttt{get}\ T; T := \texttt{del}\ T\ k; \texttt{invoke}\ k\}; Sched,$
$\quad \emptyset, \sigma[T \mapsto \{([\![e]\!], [\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle$
$\mapsto \langle \texttt{R}\ \{k := \texttt{get}\ T; T := \texttt{del}\ T\ k; \texttt{invoke}\ k\}; Sched, \emptyset\dagger, \sigma[T \mapsto \{([\![e]\!], [\texttt{skip}; \texttt{R}\ \{[\![S]\!]\}, \rho; \Pi\dagger])\}]\rangle$
$\mapsto^* \langle \texttt{R}\ \{\texttt{invoke}\ k\}; Sched, [k \mapsto [\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}, \rho; \Pi\dagger]]\dagger, \sigma[T \mapsto \emptyset]\rangle$
$\mapsto \langle \texttt{R}\ \{\texttt{R}\ \{\texttt{skip}; [\![S]\!]\}; \texttt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle$
$\mapsto \langle \texttt{R}\ \{\texttt{R}\ \{[\![S]\!]\}; \texttt{return}\}; Sched, \rho; \Pi\dagger; \emptyset\dagger, \sigma[T \mapsto \emptyset]\rangle$

6 Conclusion

In this paper, we formalized a compilation scheme for cooperative multi-tasking into delimited continuations. For the source language, we extend While with procedure calls and operations for blocking and creation of new tasks. The target language extends While with shift/reset—the target language is sequential. We then proved that the compilation scheme is correct: reductions in the source language are simulated by corresponding reductions in the target language. We have implemented this compilation scheme in our compiler from ABS to Scala. The compiler covers a much richer language than our source language, including object-oriented features, and employs the experimental continuations plugin for Scala. The compiler is integrated into the wider ABS Tool Suite, available at `http://tools.hats-project.eu/`. We are currently formalizing the results of the paper in the proof assistant Agda.

Acknowledgments This research was supported by the EU FP7 ICT project no. 231620 (HATS), the Estonian Centre of Excellence in Computer Science, EXCS, financed mainly by the European Regional Development Fund, ERDF, the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference. pp. 289–302. USENIX (2002)
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
3. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: ESOP'07 Proceedings of the 16th European conference on Programming. LNCS, vol. 4421, pp. 316–330. Springer (2007)
4. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. ACM SIGPLAN Notices - POPL'04 39(1), 123–134 (2004)
5. Danvy, O., Filinski, A.: Abstracting control. In: LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming. pp. 151–160. ACM (1990)
6. Haynes, C.T., Friedman, D.P., Wand, M.: Continuations and coroutines. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming. pp. 293–298. ACM (1984)
7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: FMCO'10 Proceedings of the 9th international conference on Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer (2012)
8. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science 365(1-2), 23–66 (2006)
9. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 11–20. ACM (2009)
10. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: generalizing active objects to concurrent components. In: ECOOP'10: Proceedings of the 24th European conference on Object-oriented programming. LNCS, vol. 6183, pp. 275–299. Springer (2010)
11. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming. LNCS, vol. 5142, pp. 104–128. Springer (2008)
12. Wand, M.: Continuation-based multiprocessing. In: LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming. pp. 19–28. ACM (1980)