Forward to a Promising Future

Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, Huu-Phuc Vo

▶ **To cite this version:**

Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, Huu-Phuc Vo. Forward to a Promising Future. 20th International Conference on Coordination Languages and Models (COORDINATION), Jun 2018, Madrid, Spain. pp.162-180, 10.1007/978-3-319-92408-3_7 . hal-01821490

HAL Id: hal-01821490
https://inria.hal.science/hal-01821490v1

Submitted on 22 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Forward to a Promising Future

Kiko Fernandez-Reyes[https://orcid.org/0000-0001-8654-118X], Dave Clarke[https://orcid.org/0000-0002-1970-6607], Elias Castegren[https://orcid.org/0000-0003-4918-6582], and Huu-Phuc Vo*

Dept. of Information Technology
Uppsala University
Uppsala, Sweden

Abstract. In many actor-based programming models,[1] asynchronous method calls communicate their results using futures, where the fulfilment occurs under-the-hood. Promises play a similar role to futures, except that they must be explicitly created and explicitly fulfilled; this makes promises more flexible than futures, though promises lack fulfilment guarantees: they can be fulfilled once, multiple times or not at all. Unfortunately, futures are too rigid to exploit many available concurrent and parallel patterns. For instance, many computations block on a future to get its result only to return that result immediately (to fulfil their own future). To make futures more flexible, we explore a construct, *forward*, that delegates the responsibility for fulfilling the current implicit future to another computation. *Forward* reduces synchronisation and gives futures promise-like capabilities. This paper presents a formalisation of the *forward* construct, defined in a high-level source language, and a compilation strategy from the high-level language to a low-level, promised-based target language. The translation is shown to preserve semantics. Based on this foundation, we describe the implementation of *forward* in the parallel, actor-based language Encore,[2] which compiles to C.

1 Introduction

Futures extend the actor programming model to express call-return synchronisation of message sends [1]. Each actor is single-threaded, but different actors execute concurrently. Communication between actors happens via asynchronous method calls (messages), which immediately return a future; futures are placeholders for the eventual result of these asynchronous method calls. An actor processes one message at a time and each message has associated a future that

* We are grateful to Joachim Parrow and Johannes Borgström for their comments regarding the bisimulation relation. We also thank the anonymous referees for their useful comments. The underlying research was funded by the Swedish VR project: SCADA.

[1] This paper focuses on futures. From this perspective we consider the actor-, task-, and active object-based models as synonymous.

[2] https://github.com/parapluu/encore

will be fulfilled with the returned value of the method. Futures are first-class values, and operations on them may be blocking, such as getting the result out of the future (`get`), or asynchronous, such as attaching a callback to a future. This last operation, known as future chaining ($f \overset{x}{\rightsquigarrow} e$), attaches a closure $\lambda x.e$ to the future f and immediately returns a new future that will contain the result of applying the closure to the value eventually stored in future f.

Consider the following code (in the actor-based language Encore [2]) that implements the broker delegation pattern: the developer's intention is to connect clients (the callers of the `Broker` actor) to a pool of actors that will actually process a job (lines 6–7):

```
1  active class Broker
2    val workers: Buffered[Worker]
3    var current: uint
4
5    def run(job: Job): int
6      val worker = this.workers[++this.current % workers.size()]
7      val future : Fut[int] = worker!start(job)
8      return get(future)
9    end
10 end
```

The problem with this code is that the connection to the `Broker` cannot be completed immediately without blocking the `Broker`'s thread of execution: returning the result of the worker running the computation requires that the `Broker` blocks until the future is fulfilled (line 8). This implementation makes the `Broker` the bottleneck of the application.

One obvious way to avoid this bottleneck is by returning the future, instead of blocking on it, as in the following code:

```
1    def run(job: Job): Fut[int]
2      val worker = this.workers[++this.current % workers.size()]
3      return worker!start(job)
4    end
```

This solution removes the blocking from `Broker`, but returns a future, which results in the client receiving a future containing a future $Fut\,(Fut\,int)$, cluttering client code and making the typing more complex.

Another way to avoid the bottleneck is to not block but yield the current thread until the future is fulfilled. This can be done using the `await` command [3, 2], which frees up the `Broker` to do other work:[3]

```
1    def run(job: Job): int
2      val worker = this.workers[++this.current % workers.size()]
3      val future = worker!start(job)
4      await(future)
5      return get(future)
6    end
```

This solution frees up the `Broker`, but can result in a lot of memory being consumed to hold the waiting instances of calls `Broker.run()`.

[3] The essential difference between `get` and `await` is that `get` blocks an actor, whereas `await` blocks only the current method invocation and frees up the actor.

Another alternative is to use promises [4]. A promise can be passed around and fulfilled explicitly at the point where the corresponding result is known. Passing a promise around is akin to passing the responsibility to provide a particular result, thereby fulfilling the promise.

```
1   def run(job: Job, promise: Promise[int]): unit
2     val worker = this.workers[++this.current % workers.size()]
3     worker!start(job, promise)
4   end
5
6  class Worker
7   def start(job: Job, promise: Promise[int]) : unit
8     // actually do job
9     promise.fulfil(result)
10  end
11 end
```

Promises are problematic because they diverge from the commonplace call-return control flow, there is no explicit requirement to actually fulfil a promise, and care is required to avoid fulfilling multiple times. This latter issue, fulfilling a promise multiple times, can be solved by a substructural type system, which guarantees a single writer to the promise [5, 6]. Substructural type systems are more complex and not mainstream, which rules out adoption in languages such as Java and C#. Our solution relies on futures and is suitable for mainstream languages.

The main difference between promises and futures are that developers *explictly* create and fulfil promises, whereas futures are *implictly* created and fulfilled. Promises are thus more flexible at the expense of any fulfilment guarantees.

This paper explores a construct called *forward* that retains the guarantees of using futures, while allowing some degree of delegation of responsibility to fulfil a future, as in promises. This construct was first proposed a while ago [7], but only recently has been implemented in the language Encore [2].

With *forward*, the `run` of `Broker` method now becomes:

```
1   def run(job: Job): int
2     val worker = this.workers[++this.current % workers.size()]
3     forward(worker!start(job))
4   end
```

Forward delegates the fulfilment of the future that `run` will put its result in, to the call `worker!start(job)`. Using *forward* frees up the `Broker` object, as `run` completes immediately, though the future is fulfilled only when `worker!start(job)` produces a result.

The paper makes the following contributions:

- a formalisation and soundness proof of the *forward* construct in a concise, high-level language (Section 2);
- a formalisation of a low-level, promise-based language (Section 3),
- a translation from the high-level language to the low-level language, a proof of program equivalence, between the high-level language and its translation to the low-level language (Section 4); and
- microbenchmarks that compare the `get`-and-`return` and `await`-and-`get` pattern versus the `forward` construct (Section 5).

2 A Core Calculus of Futures and Forward

This section presents a core calculus that includes tasks, futures and operations on them, and forward. The calculus consists of two levels: expressions and configurations. Expressions correspond to programs and what tasks evaluate. Configurations capture the run-time configuration; they are collections of tasks (running expressions), futures, and chains. This calculus is much more concise than the previous formalisation of forward [7].

The syntax of the core calculus is as follows:

$$e ::= v \mid e\ e \mid \texttt{async}\ e \mid e \overset{x}{\rightsquigarrow} e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{forward}\ e \mid \texttt{get}\ e$$
$$v ::= c \mid f \mid x \mid \lambda x.e$$

Expressions include values (v), function application ($e\ e$), spawning asynchronous computations ($\texttt{async}\ e$), future chaining ($e \overset{x}{\rightsquigarrow} e'$), which attaches $\lambda x.e'$ onto a future to run as soon as the future produced by e is fulfilled, *if-then-else* expressions, `forward`, and `get`, which extracts the value from a future. Values are constants (c), futures (f), variables (x) and lambda abstractions ($\lambda x.e$). The calculus has neither actors nor message sends/method calls. For our purposes, tasks play the role of actors and spawning asynchronous computations is analogous to message sends.

Configurations, *config*, give a partial view on the system and are (non-empty) multisets of tasks, futures and chains. They have the following syntax:

$$\mathit{config} ::= (\mathit{fut}_f) \mid (\mathit{fut}_f\ v) \mid (\mathit{task}_f\ e) \mid (\mathit{chain}_f\ f\ e) \mid \mathit{config}\ \mathit{config}$$

Future configurations are (fut_f) and $(\mathit{fut}_f\ v)$, representing an unfulfilled future f and a fulfilled future f with value v. Configuration $(\mathit{task}_f\ e)$ is a task running expression e that will write the result of e in future f.[4] Configuration $(\mathit{chain}_f\ g\ e)$ denotes a computation that waits until future g is fulfilled, applies expression e to the value stored in g in a new task whose result will be stored in future f.

The initial configuration for program e is $(\mathit{task}_f\ e)\ (\mathit{fut}_f)$, where the result of e will be written into future f at the end result of the program's execution.

2.1 Operational Semantics

The operational semantics use a small-step semantics with reduction-based, contextual rules for evaluation within tasks. Evaluation contexts E contains a hole $\bullet$ that denotes where the next reduction step happens [8]:

$$E ::= \bullet \mid E\ e \mid v\ E \mid E \overset{x}{\rightsquigarrow} e \mid \texttt{forward}\ E \mid \texttt{get}\ E \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e$$

[4] A reviewer suggested that (fut_f), $(\mathit{fut}_f\ v)$, and $(\mathit{task}_f\ e)$ could be combined into a single configuration component. We have considered this conflation in the past. While it would reduce the complexity of the calculus, it would also make compilation into the target calculus and the proofs of correctness more complex.

$$\frac{}{(task_f\ E[\texttt{if}\ true\ \texttt{then}\ e\ \texttt{else}\ e'])\to(task_f\ E[e])}\ (\textsc{Red-If-True}) \qquad \frac{}{(task_f\ E[\lambda x.e\ v])\to(task_f\ E[e[v/x]])}\ (\textsc{Red-}\beta)$$

$$\frac{}{(task_f\ E[\texttt{if}\ false\ \texttt{then}\ e\ \texttt{else}\ e'])\to(task_f\ E[e'])}\ (\textsc{Red-If-False}) \qquad \frac{}{(task_f\ E[\texttt{forward}\ h])\to(chain_f\ h\ \lambda x.x)}\ (\textsc{Red-Fwd-Fut})$$

$$\frac{}{(chain_g\ f\ e)\ (fut_f\ v)\to(task_g\ (e\ v))\ (fut_f\ v)}\ (\textsc{Red-Chain-Run}) \qquad \frac{}{(task_f\ E[\texttt{get}\ h])\ (fut_h\ v)\to(task_f\ E[v])\ (fut_h\ v)}\ (\textsc{Red-Get})$$

$$\frac{}{(task_f\ v)\ (fut_f)\to(fut_f\ v)}\ (\textsc{Red-Fut-Fulfil}) \qquad \frac{fresh\ f}{(task_g\ E[\texttt{async}\ e])\to(fut_f)\ (task_f\ e)\ (task_g\ E[f])}\ (\textsc{Red-Async})$$

$$\frac{fresh\ g}{(task_f\ E[h\overset{x}{\rightsquigarrow}e])\to(fut_g)\ (chain_g\ h\ \lambda x.e)\ (task_f\ E[g])}\ (\textsc{Red-Chain-Create})$$

Fig. 1: Reduction Rules. f, g, h range over futures.

$$\frac{config\to config''}{config\ config'\to config''\ config'} \qquad \frac{config\equiv config' \quad config'\to config'' \quad config''\equiv config'''}{config\to config'''}$$

Fig. 2: Configuration evaluation rules. Equivalence $\equiv$ (omitted) captures the fact that configurations are a multiset of basic configurations.

The evaluation rules are given in Fig. 1. The evaluation of *if-then-else* expressions and functions applications proceed in the standard fashion (Red-If-True, Red-If-False, and Red-β). The *async* construct spawns a new task to execute the given expression, and creates a new future to store its result (Red-Async). When the spawned task finishes its execution, it places the value in the designated future (Red-Fut-Fulfil). To obtain the contents of a future, the blocking construct *get* stops the execution of the task until the future is fulfilled (Red-Get). Chaining an expression on a future results immediately in a new future that will eventually contain the result of evaluating the expression, and a chain configuration storing the expression is connected with the original future (Red-Chain-Create). When the future is fulfilled, any chain configurations become task configurations and start evaluating the stored expression on the value stored in the future (Red-Chain-Run). Forward applies to a future where the result of the future computation will be the result of the current computation, stored in the future associated with the current task. Forwarding to future h throws away the remainder of the body of the current task and chains the identity function on the future, the effect of which is to copy the eventual result stored in h into the current future (Red-Fwd-Fut).

The configuration evaluation rules (Fig. 2) describe how configurations make progress, which is either by some subconfiguration making progress, or by rewriting a configuration to one that will make progress using the equations of multisets.

Example and Optimisations The following example illustrates some aspects of the calculus.

$$
\begin{aligned}
&(task_f\ E[\texttt{async}\ \ (\texttt{forward}\ h)])\ (fut_h\ 42)\\
&\xrightarrow{\textsc{Red-Async}}\ \ (task_f\ E[g])\ (fut_h\ 42)\ (fut_g)\ (task_g\ \texttt{forward}\ h)\\
&\xrightarrow{\textsc{Red-Fwd-Fut}}\ \ (task_f\ E[g])\ (fut_h\ 42)\ (fut_g)\ (chain_g\ h\ \lambda x.x)\\
&\xrightarrow{\textsc{Red-Chain-Run}}\ \ (task_f\ E[g])\ (fut_h\ 42)\ (fut_g)\ (task_g\ (\lambda x.x)\ 42)\\
&\xrightarrow{\textsc{Red-}\beta}\ \ (task_f\ E[g])\ (fut_h\ 42)\ (fut_g)\ (task_g\ 42)\\
&\xrightarrow{\textsc{Red-Fut-Fulfil}}\ \ (task_f\ E[g])\ (fut_h\ 42)\ (fut_g\ 42)
\end{aligned}
$$

Firstly, a new task is spawned with the use of `async`. This task forwards the responsibility to fulfil its future to (the task fulfilling) future h, i.e. future g gets fulfilled with the value contained in future h.

Two special cases of `forward` can be given more direct reduction sequences, which correspond to optimisations performed in the Encore compiler. The first case corresponds to forwarding directly to another method call, which is the primary use case for `forward`, namely, forwarding to another method `forward(e!m())`. The optimised reduction rule is

$$(task_f\ E[\texttt{forward}\,(\texttt{async}\ e)]) \to (task_f\ e)$$

For comparison, the standard reduction sequence[5] is

$$
\begin{aligned}
&(task_f\ E[\texttt{forward}\ \ (\texttt{async}\ e)]) \to (task_f\ E[\texttt{forward}\ g])\ (task_g\ e)\ (fut_g)\\
&\to (chain_f\ g\ \lambda x.x)\ (task_g\ e)\ (fut_g) \to^* (chain_f\ g\ \lambda x.x)\ (task_g\ v)\ (fut_g)\\
&\to (chain_f\ g\ \lambda x.x)\ (fut_g\ v) \to (task_f\ (\lambda x.x)\ v)\ (fut_g\ v) \to (task_f\ v)\ (fut_g\ v)
\end{aligned}
$$

This can be seen as equivalent to the reduction sequence

$$(task_f\ E[\texttt{forward}\,(\texttt{async}\ e)]) \to (task_f\ e) \to^* (task_f\ v)$$

because the future g will no longer be accessible.

Similarly, forwarding a future chain can be reduced directly to a chain configuration:

$$(task_f\ E[\texttt{forward}\,(h \overset{x}{\rightsquigarrow} e)]) \to (chain_f\ h\ \lambda x.e)$$

In both cases, forward can be seen as making a call-with-current-future.

[5] $\to^*$ is the reflexive, transitive closure of the reduction relation $\to$.

$$\frac{\text{c is a constant of type } \tau}{\Gamma \vdash_\rho c : \tau}\ (\text{T-Constant}) \qquad \frac{f : \mathit{Fut}\ \tau \in \Gamma}{\Gamma \vdash_\rho f : \mathit{Fut}\ \tau}\ (\text{T-Future}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_\rho x : \tau}\ (\text{T-Variable}) \qquad \frac{\Gamma, x : \tau \vdash_\bullet e : \tau'}{\Gamma \vdash_\rho \lambda x.e : \tau \to \tau'}\ (\text{T-Abstraction})$$

$$\frac{\Gamma \vdash_\rho e_1 : \tau \to \tau' \quad \Gamma \vdash_\rho e_2 : \tau}{\Gamma \vdash_\rho e_1\ e_2 : \tau'}\ (\text{T-Application}) \qquad \frac{\Gamma \vdash_\rho e : \mathit{bool} \quad \Gamma \vdash_\rho e' : \tau \quad \Gamma \vdash_\rho e'' : \tau}{\Gamma \vdash_\rho \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' : \tau}\ (\text{T-If-Then-Else}) \qquad \frac{\Gamma \vdash_\rho e : \mathit{Fut}\ \tau}{\Gamma \vdash_\rho \texttt{get}\ e : \tau}\ (\text{T-Get})$$

$$\frac{\Gamma \vdash_\tau e : \tau}{\Gamma \vdash_\rho \texttt{async}\ e : \mathit{Fut}\ \tau}\ (\text{T-Async}) \qquad \frac{\Gamma \vdash_\rho e : \mathit{Fut}\ \tau \quad \Gamma, x : \tau \vdash_{\tau'} e' : \tau'}{\Gamma \vdash_\rho e \overset{x}{\rightsquigarrow} e' : \mathit{Fut}\ \tau'}\ (\text{T-Chain}) \qquad \frac{\Gamma \vdash_\rho e : \mathit{Fut}\ \rho \quad \rho \neq \bullet}{\Gamma \vdash_\rho \texttt{forward}\ e : \tau}\ (\text{T-Forward})$$

Fig. 3: Typing Rules

2.2 Static Semantics

The type system has basic types, K, and future types:

$$\tau ::= K \mid \mathit{Fut}\ \tau$$

The typing rules (Fig. 3) define the judgement $\Gamma \vdash_\rho e : \tau$, which states that in the typing environment Γ, which gives the types of futures and free variables, expression e has type τ, where ρ is the *expected task type*, the result type of the task in which the expression appears. ρ ranges over both types τ and symbol $\bullet$ which is not a type. $\bullet$ is used to prevent the use of `forward` in contexts where the expected task type is not clear, specifically within closures, as a closure can be passed between tasks and run in a context different from their defining contexts. The types of constants are assumed to be provided (Rule T-Constant). Variables and futures types are defined in the typing environment (Rules T-Variable and T-Future). Function application and abstraction have the standard typing rules (Rules T-Application and T-Abstraction), except that within the body of a closure the expected task type is not known. When `async` is applied to an expression e, a new task is created and the expected task type changes to the type of the expression. The result type of the `async` call is a future type of the expression's type (Rule T-Async). Chaining is essentially mapping for the *Fut* type constructor, and rule T-Chain reflects this fact. In addition, because chaining ultimately creates a new task to run the expression, the expected task type ρ changes to the return type of the expression. Getting the value from a future of some type results in a value of that type (Rule T-Get). Forwarding requires the argument to `forward` to be a future of the same type as the expected task type (Rule T-Forward). As `forward` does not return locally, the result type is arbitrary.

Well-formed configurations, $\Gamma \vdash \mathit{config\ ok}$, are typed against environment, Γ, that gives the types of futures (Fig. 4). The type rules depend on the following definitions.

$$\frac{f \in dom(\Gamma)}{\Gamma \vdash (fut_f)\ ok}\ (\textsc{Fut}) \qquad \frac{f : Fut\ \tau \in \Gamma \quad \Gamma \vdash_\bullet v : \tau}{\Gamma \vdash (fut_f\ v)\ ok}\ (\textsc{F-Fut}) \qquad \frac{f : Fut\ \tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (task_f\ e)\ ok}\ (\textsc{Task})$$

$$\frac{f : Fut\ \tau \in \Gamma \quad g : Fut\ \tau' \in \Gamma \quad \Gamma \vdash_\tau e : \tau' \to \tau}{\Gamma \vdash (chain_f\ g\ e)\ ok}\ (\textsc{Chain})$$

$$\frac{\Gamma \vdash config_1\, ok \quad \Gamma \vdash config_2\, ok \quad defs(config_1) \cap defs(config_2) = \varnothing \quad writers(config_1) \cap writers(config_2) = \varnothing}{\Gamma \vdash config_1\ config_2\ ok}\ (\textsc{Config})$$

Fig. 4: Configuration typing

Definition 1. *The function defs(config) extracts the set of futures present in a configuration config.*

$$\begin{aligned} &defs((fut_f)) = defs((fut_f\ v)) = \{f\} \\ &defs((config_1\ config_2)) = defs(config_1) \cup defs(config_2) \\ &defs(\_) = \varnothing \end{aligned}$$

Definition 2. *The function writers(config) extracts the set of writers to futures in configuration config.*

$$\begin{aligned} &writers((chain_f\ g\ e)) = writers((task_f\ e)) = \{f\} \\ &writers(config_1\ config_2) = writers(config_1) \cup writers(config_2) \\ &writers(\_) = \varnothing \end{aligned}$$

Rules Fut and F-Fut define well-formed future configurations. Rules Task and Chain define well-formed task and future chaining configurations and set the expected task types. Rule Config defines how to build larger configurations from smaller ones. Each future may be defined at most once and there is at most one writer to each future.

The rules for well-formed configurations apply to partial configurations. Complete configurations can be typed by adding extra conditions to ensure that all futures in Γ have a future configuration, there is a one-to-one correspondence between tasks/chains and unfulfilled futures, and dependencies between tasks are acyclic. These definitions have been omitted and are similar to those found in our earlier work [9].

Formal Properties The proof of soundness of the type system follows standard techniques [8]. The proof of progress requires that there is no deadlock, which follows as there is no cyclic dependency between tasks [9].

Lemma 1 (Type preservation). *If $\Gamma \vdash config\ ok$ and $config \to config'$, then there exists a Γ' such that $\Gamma' \supset \Gamma$ and $\Gamma' \vdash config'\ ok$*

Proof. By induction on the derivation of config → config′. □

Definition 3 (Terminal Configuration). *A complete configuration config is terminal iff every element of the configuration has the shape:* $(fut_f\ v)$.

Lemma 2 (Progress). *For a complete configuration config, if* $\Gamma \vdash$ *config ok, then config is a terminal configuration or there exists a config′ such that config* $\rightarrow$ *config′.*

Proof. By induction on a derivation of config → config′, relying on the invariance of the acyclicity of task dependencies. □

3 A Promising Implementation Calculus

The implementation of *forward* in the Encore programming language is via compilation into C, linking with Pony's actor-based run-time [10]. At this level, Encore's futures are treated like promises in that they are passed around to the place where the result of a method call is known in order to be fulfilled. To model this implementation approach, we introduce a low-level target calculus based on tasks and promises. This section presents the formalised target calculus, and the next section presents the compilation strategy from the source to the target language.

The syntax of the target language is as follows:

$$
\begin{aligned}
e &::= v \mid e\ e \mid \texttt{Task}(e, e) \mid \texttt{stop} \mid e; e \mid \texttt{Prom} \mid \texttt{fulfil}(e, e) \mid \texttt{get}\ e \\
&\quad \mid \texttt{Chain}(e, e, e) \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
v &::= c \mid f \mid x \mid \lambda x.e \mid ()
\end{aligned}
$$

Expressions consist of values, function application $(e\ e)$, sequential composition of expressions $(e; e)$, the spawning and stopping of tasks ($\texttt{Task}(e, e)$ and $\texttt{stop}$), the creation, fulfilment, reading, and chaining of promises ($\texttt{Prom}$, $\texttt{fulfil}(e, e)$, $\texttt{get}\ e$, and $\texttt{Chain}(e, e, e)$) and the standard *if-then-else* expression. Values are constants, futures, variables, abstractions and unit (). The main differences with the source language are that tasks have to be explicitly stopped, which captures non-local exit, and promises must be explicitly created and fulfilled.

3.1 Operational Semantics

The semantics of the target calculus is analogous to the source calculus. The evaluation contexts are:

$$
\begin{aligned}
E ::= \bullet &\mid E\ e \mid v\ E \mid E; e \mid \texttt{get}\ E \mid \texttt{fulfil}(E, e) \mid \texttt{fulfil}(v, E) \\
&\mid \texttt{Task}(E, e) \mid \texttt{Chain}(e, E, e) \mid \texttt{Chain}(E, v, e) \mid \texttt{Chain}(v, v, E) \\
&\mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e
\end{aligned}
$$

$$\frac{}{(\texttt{task}\ E[\texttt{if}\ \mathit{true}\ \texttt{then}\ e\ \texttt{else}\ e'])\rightarrow(\texttt{task}\ E[e])}\ (\text{RI-If-True})$$

$$\frac{}{(prm_f\ v)\ (\texttt{task}\ E[\texttt{fulfil}(f,v')])\rightarrow\texttt{ERROR}}\ (\text{RI-ERROR})$$

$$\frac{}{(\texttt{task}\ E[\texttt{if}\ \mathit{false}\ \texttt{then}\ e\ \texttt{else}\ e'])\rightarrow(\texttt{task}\ E[e'])}\ (\text{RI-If-False})$$

$$\frac{\mathit{fresh}\,f}{(\texttt{task}\ E[\texttt{Prom}])\rightarrow(prm_f)\ (\texttt{task}\ E[f])}\ (\text{RI-Promise})$$

$$\frac{}{(\texttt{task}\ E[v;e])\rightarrow(\texttt{task}\ E[e])}\ (\text{RI-Statement})$$

$$\frac{}{(\texttt{task}\ E[\texttt{Chain}(f,g,(\lambda x.e))])\rightarrow(\texttt{chain}\ g\ e[f/x])\ (\texttt{task}\ E[f])}\ (\text{RI-Chain})$$

$$\frac{}{(\texttt{task}\ E[(\lambda x.e)\ v])\rightarrow(\texttt{task}\ E[e[v/x]])}\ (\text{RI-}\beta)$$

$$\frac{}{(prm_f)\ (\texttt{task}\ E[\texttt{fulfil}(f,v)])\rightarrow(prm_f\ v)\ (\texttt{task}\ E[()])}\ (\text{RI-Fulfil})$$

$$\frac{}{(\texttt{task}\ E[\texttt{stop}])\rightarrow\epsilon}\ (\text{RI-Stop})$$

$$\frac{}{(\texttt{task}\ E[\texttt{Task}(f,(\lambda x.e))])\rightarrow(\texttt{task}\ E[f])\ (\texttt{task}\ e[f/x])}\ (\text{RI-Task})$$

$$\frac{}{(\texttt{chain}\ g\ e)\ (prm_g\ v)\rightarrow(\texttt{task}\ (e\ v))\ (prm_g\ v)}\ (\text{RI-Config-Chain})$$

$$\frac{}{(\texttt{task}\ E[\texttt{get}\ h])\ (prm_h\ v)\rightarrow(\texttt{task}\ E[v])\ (prm_h\ v)}\ (\text{RI-Get})$$

Fig. 5: Target reduction rules

Configurations are multisets of promises, tasks, and chains:

$$\mathit{config} ::= \epsilon \mid (prm_f) \mid (prm_f\ v) \mid (\texttt{task}\ e) \mid (\texttt{chain}\ f\ e) \mid \mathit{config}\ \mathit{config}$$

The empty configuration is represented by ϵ, an unfulfilled promise is written as (prm_f) and a fulfilled promise holding value v is written as $(prm_f\ v)$.

Tasks and chains work in the same way as in the source language, except that they work now on promises (Fig. 5). Promises are handled much more explicitly than futures are, and need to be passed around like regular values. The creation of a task needs a promise and a function to run; the spawned task runs the function, has access to the passed promise and leaves the promise reference in the spawning task (RI-Task). Stopping a task just finishes the task (RI-Stop). The construct `Prom` creates an empty promise (RI-Promise). Fulfilling a promise results in the value being stored if the promise was empty (RI-Fulfil), or an error otherwise (RI-Error). Promises are chained in a similar fashion to futures: the construct $\texttt{Chain}(f,g,e)$ immediately passes the promise f to expression e — the intention being that f will hold the eventual result; the chain then waits on promise g, and passes the value it receives into expression $(e\ f)$ (RI-Chain and RI-Config-Chain). The target language borrows the configuration evaluation rules from the source language (Fig. 2).

Example For illustration purposes we translate the example from the high-level language, $(fut_f)\ (task_f\ E[\texttt{forward}\,(\texttt{async}\ e)])$ shown in Section 2, and show the reduction steps of the low-level language:

(prm_f) (task $E[\texttt{Chain}(f, \texttt{Task}(\texttt{Prom}, (\lambda \texttt{d}'.\texttt{fulfil}(\texttt{d}', e); \texttt{stop})), \lambda \texttt{d}'.\lambda x.\texttt{fulfil}(\texttt{d}', x); \texttt{stop}); \texttt{stop}]$)

$\longrightarrow (prm_f)\ (prm_g)$ (task $E[\texttt{Chain}(f, \texttt{Task}(g, (\lambda \texttt{d}'.\texttt{fulfil}(\texttt{d}', e); \texttt{stop})), \lambda \texttt{d}'.\lambda x.\texttt{fulfil}(\texttt{d}', x); \texttt{stop}); \texttt{stop}]$)

$\longrightarrow (prm_f)\ (prm_g)$ (task $E[\texttt{Chain}(f, g, \lambda \texttt{d}'.\lambda x.\texttt{fulfil}(\texttt{d}', x); \texttt{stop}); \texttt{stop}]$) (task $\texttt{fulfil}(g, e); \texttt{stop}$)

$\longrightarrow (prm_f)\ (prm_g)$ (task $E[f; \texttt{stop}]$) (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task $\texttt{fulfil}(g, e); \texttt{stop}$)

$\longrightarrow (prm_f)\ (prm_g)$ (task $E[\texttt{stop}]$) (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task $\texttt{fulfil}(g, e); \texttt{stop}$)

$\longrightarrow (prm_f)\ (prm_g)$ (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task $\texttt{fulfil}(g, e); \texttt{stop}$)

$\longrightarrow^* (prm_f)\ (prm_g)$ (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task $\texttt{fulfil}(g, v); \texttt{stop}$)

$\longrightarrow (prm_f)\ (prm_g\ v)$ (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task $(); \texttt{stop}$)

$\longrightarrow (prm_f)\ (prm_g\ v)$ (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$) (task stop)

$\longrightarrow (prm_f)\ (prm_g\ v)$ (chain g $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})$)

$\longrightarrow (prm_f)\ (prm_g\ v)$ (task $(\lambda x.\texttt{fulfil}(f, x); \texttt{stop})\ v$)

$\longrightarrow (prm_f)\ (prm_g\ v)$ (task $\texttt{fulfil}(f, v); \texttt{stop}$)

$\longrightarrow (prm_f\ v)\ (prm_g\ v)$ (task $(); \texttt{stop}$)

$\longrightarrow (prm_f\ v)\ (prm_g\ v)$ (task stop)

$\longrightarrow (prm_f\ v)\ (prm_g\ v)$

We show how the compilation strategy proceeds in Section 4.

3.2 Static Semantics

The type system has basic types, K, and promise types defined below:

$$\tau ::= K \mid \mathit{Prom}\ \tau$$

The type rules define the judgment $\Gamma \vdash e : \tau$ which states that, in the environment Γ, which records the types of promises and free variables, expression e has type τ. The rules for constants, promises, and variables, *if-then-else*, abstraction and function application are analogous to the source calculus, except no expected task type is recorded. The unit value has type `unit` (TI-Unit); the `stop` expression finishes a task and has any type (TI-Stop). The creation of a promise has type $\mathit{Prom}\ \tau$ (TI-Promise-New); the fulfilment of a promise $\texttt{fulfil}(e, e')$ has type `unit` and requires the first parameter to be a promise and the second to be an expression that matches the type of the promise (TI-Fulfil). To spawn a task ($\texttt{Task}(e, e)$), the first argument of the task must be a promise and the second a function that takes a promise having the same type as the first argument (TI-Task); promises can be chained on with functions that run if the promise is fulfilled: $\texttt{Chain}(e, e', e'')$ has type $\mathit{Prom}\ \tau$ and e and e' are promises and e'' is an abstraction that takes arguments of the first and second promise types. Both task and chain constructors return the promise that is passed to them, for convenience in the compilation scheme.

Soundness of the type system is proven using standard techniques.

$$\frac{\text{c is a constant of type } \tau}{\Gamma \vdash c : \tau}\ (\text{TI-Constant}) \qquad \frac{f\ : \mathit{Prom}\ \tau \in \Gamma}{\Gamma \vdash f : \mathit{Prom}\ \tau}\ (\text{TI-Promise}) \qquad \frac{x\ :\ \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{TI-Variable}) \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}\ (\text{TI-Unit})$$

$$\frac{}{\Gamma \vdash \texttt{stop} : \tau}\ (\text{TI-Stop}) \qquad \frac{}{\Gamma \vdash \texttt{Prom} : \mathit{Prom}\ \tau}\ (\text{TI-Promise-New}) \qquad \frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash e'' : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' : \tau}\ (\text{TI-If})$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau}\ (\text{TI-Statement}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ (\text{TI-Abstraction}) \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}\ (\text{TI-App})$$

$$\frac{\Gamma \vdash e : \mathit{Prom}\ \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \texttt{fulfil}(e, e') : \texttt{unit}}\ (\text{TI-Fulfil}) \qquad \frac{\Gamma \vdash e : \mathit{Prom}\ \tau \quad \Gamma \vdash e' : \mathit{Prom}\ \tau \to\ \tau'}{\Gamma \vdash \texttt{Task}(e, e') : \mathit{Prom}\ \tau}\ (\text{TI-Task})$$

$$\frac{\Gamma \vdash e : \mathit{Prom}\ \tau}{\Gamma \vdash \texttt{get}\ e : \tau}\ (\text{TI-Get}) \qquad \frac{\Gamma \vdash e : \mathit{Prom}\ \tau \quad \Gamma \vdash e' : \mathit{Prom}\ \tau' \quad \Gamma \vdash e'' : \mathit{Prom}\ \tau \to \tau' \to\ \tau''}{\Gamma \vdash \texttt{Chain}(e, e', e'') : \mathit{Prom}\ \tau}\ (\text{TI-Chain})$$

$$\frac{f \in \mathit{dom}(\Gamma)}{\Gamma \vdash (\mathit{prm}_f)\ \mathit{ok}}\ (\text{Prom}) \qquad \frac{f : \mathit{Prom}\ \tau \in \Gamma}{\Gamma \vdash (\mathit{prm}_f\ v)\ \mathit{ok}}\ (\text{F-Prom}) \qquad \frac{\Gamma \vdash f : \mathit{Prom}\ \tau \qquad \Gamma \vdash e : \tau \to \tau''}{\Gamma \vdash (\texttt{chain}\ f\ e)\ \mathit{ok}}\ (\text{Chain-Target})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{task}\ e)\ \mathit{ok}}\ (\text{Task-Target}) \qquad \frac{\Gamma \vdash \mathit{config}_1\ \mathit{ok} \qquad \Gamma \vdash \mathit{config}_2\ \mathit{ok}}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \mathit{ok}}\ (\text{Config-Target})$$

Fig. 6: Typing rules for expressions and configurations in the target language

4 Compilation: From Futures and Forward to Promises

This section presents the compilation function from the source to the target language and outlines a proof that it preserves semantics. The compilation strategy is defined inductively (Fig. 7); the compilation of expressions, denoted $\mathcal{C}[\![e]\!]_{\texttt{destiny}}$, takes an expression e and a meta-variable `destiny` which holds the promise that the current task should fulfil, and produces an expression in the target language.

Futures are translated to promises, and most other expressions are translated homomorphically. The constructs where something interesting happens are `async`, `forward` and future chaining; these constructs adopt a common pattern implemented using a two parameter lambda abstraction: the first parameter, variable $\texttt{destiny}'$, is the promise to be fulfilled and the second parameter is the value that fulfils the promise. The best illustration of how `forward` behaves differently from a regular asynchronous call is the difference in the rules

$\boxed{\mathcal{C}[\![e]\!]_{\texttt{destiny}}}$ Compilation Strategy

$$\mathcal{C}[\![f]\!]_{\texttt{destiny}} = f \quad \mathcal{C}[\![x]\!]_{\texttt{destiny}} = x \quad \mathcal{C}[\![c]\!]_{\texttt{destiny}} = c$$
$$\mathcal{C}[\![\lambda x.e]\!]_{\texttt{destiny}} = \lambda x.\mathcal{C}[\![e]\!]_{\texttt{destiny}}$$
$$\mathcal{C}[\![e_1\ e_2]\!]_{\texttt{destiny}} = \mathcal{C}[\![e_1]\!]_{\texttt{destiny}}\ \mathcal{C}[\![e_2]\!]_{\texttt{destiny}}$$
$$\mathcal{C}[\![\texttt{get}\ e]\!]_{\texttt{destiny}} = \texttt{get}\ \mathcal{C}[\![e]\!]_{\texttt{destiny}}$$
$$\mathcal{C}[\![\texttt{async}\ e]\!]_{\texttt{destiny}} = \texttt{Task}(\texttt{Prom}, (\lambda\texttt{destiny}'.\texttt{fulfil}(\texttt{destiny}', \mathcal{C}[\![e]\!]_{\texttt{destiny}'}); \texttt{stop}))$$
$$\mathcal{C}[\![\texttt{forward}\ e]\!]_{\texttt{destiny}} = \texttt{Chain}(\texttt{destiny}, \mathcal{C}[\![e]\!]_{\texttt{destiny}}, \lambda\texttt{destiny}'.\lambda x.\texttt{fulfil}(\texttt{destiny}', x); \texttt{stop}); \texttt{stop}$$
$$\mathcal{C}[\![e \overset{x}{\rightsquigarrow} e']\!]_{\texttt{destiny}} = \texttt{Chain}(\texttt{Prom}, \mathcal{C}[\![e]\!]_{\texttt{destiny}}, (\lambda\texttt{destiny}'.\lambda x.\texttt{fulfil}(\texttt{destiny}', \mathcal{C}[\![e']\!]_{\texttt{destiny}'}); \texttt{stop}))$$

$\boxed{\mathcal{C}[\![e]\!]_{\texttt{destiny}}}$ Optimised Compilation Strategy

$$\mathcal{C}[\![\texttt{forward}(\texttt{async}(e))]\!]_{\texttt{destiny}} =$$
$$\texttt{Task}(\texttt{destiny}, (\lambda\texttt{destiny}'.\texttt{fulfil}(\texttt{destiny}', \mathcal{C}[\![e]\!]_{\texttt{destiny}'}); \texttt{stop})); \texttt{stop}$$
$$\mathcal{C}[\![\texttt{forward}\ (e \overset{x}{\rightsquigarrow} e')]\!]_{\texttt{destiny}} =$$
$$\texttt{Chain}(\texttt{destiny}, \mathcal{C}[\![e]\!]_{\texttt{destiny}}, \lambda\texttt{destiny}'.\lambda x.\texttt{fulfil}(\texttt{destiny}', \mathcal{C}[\![e']\!]_{\texttt{destiny}'}); \texttt{stop}); \texttt{stop}$$

$\boxed{\mathcal{T}[\![\mathit{config}]\!]}$ Configuration Compilation Strategy

$$\mathcal{T}[\![(\mathit{fut}_f)]\!] = (\mathit{prm}_f) \qquad \mathcal{T}[\![(\mathit{task}_f\ e)]\!] = (\texttt{task}\ \texttt{fulfil}(f, \mathcal{C}[\![e]\!]_f); \texttt{stop})$$
$$\mathcal{T}[\![(\mathit{fut}_f\ v)]\!] = (\mathit{prm}_f\ \mathcal{C}[\![v]\!]_f) \quad \mathcal{T}[\![\mathit{config}\ \mathit{config}']\!] = \mathcal{T}[\![\mathit{config}]\!]\ \mathcal{T}[\![\mathit{config}']\!]$$
$$\mathcal{T}[\![(\mathit{chain}_f\ g\ e)]\!] = (\texttt{chain}\ g\ (\lambda x.\texttt{fulfil}(f, \mathcal{C}[\![e]\!]_f\ x); \texttt{stop}))$$
where x is fresh
$$\mathcal{T}[\![(\mathit{task}_f\ e)]\!] = (\texttt{task}\ (\texttt{fulfil}(f, \mathcal{C}[\![e]\!]_f); \texttt{stop}))$$

$\boxed{\mathcal{C}[\![\tau]\!]}$ Type translation

$$\mathcal{C}[\![\mathit{ok}]\!] = \mathit{ok} \qquad \mathcal{C}[\![K]\!] = K$$
$$\mathcal{C}[\![\mathit{Fut}\,\tau]\!] = \mathit{Prom}\ \ \mathcal{C}[\![\tau]\!] \qquad \mathcal{C}[\![\tau \to \tau']\!] = \mathcal{C}[\![\tau]\!] \to \mathcal{C}[\![\tau']\!]$$

$\boxed{\mathcal{T}[\![\Gamma \vdash f : \tau]\!]}$ Environment Translation

$$\mathcal{T}[\![\Gamma \vdash_\rho \mathit{config}]\!] = \mathcal{C}[\![\Gamma]\!] \vdash \mathcal{T}[\![\mathit{config}]\!] \qquad \mathcal{C}[\![\varnothing]\!] = \epsilon$$
$$\mathcal{C}[\![\Gamma, f : \mathit{Fut}\ \tau]\!] = \mathcal{C}[\![\Gamma]\!], \mathcal{C}[\![f : \mathit{Fut}\ \tau]\!] \qquad \mathcal{C}[\![x : \tau]\!] = x : \mathcal{C}[\![\tau]\!]$$
$$\mathcal{C}[\![\Gamma, x : \tau]\!] = \mathcal{C}[\![\Gamma]\!], \mathcal{C}[\![x : \tau]\!] \qquad \mathcal{C}[\![f : \mathit{Fut}\ \tau]\!] = f : \mathcal{C}[\![\mathit{Fut}\ \tau]\!]$$

Fig. 7: Compilation strategy of terms, configurations, types and typing rules

for `async` e and the optimised rule for `forward` (`async` e). The translation of `async` e creates a new promise to store e's result value, whereas the translation of `forward`(`async` e) reuses the promise from the context, namely the one passed in via the `destiny` variable.

The compilation of configurations, denoted $\mathcal{T}[\![\mathit{config}]\!]$, translates configurations from the source language to the target language. For example, the compilation of the source configuration (fut_f) $(\mathit{task}_f$ `forward` (`async` e)) compiles into:

$$\begin{aligned}\mathcal{T}[\![(\mathit{fut}_f)\ (\mathit{task}_f\ \texttt{forward}\ (\texttt{async}\ e))]\!] = \\ \mathcal{T}[\![(\mathit{fut}_f)]\!]\ \mathcal{T}[\![(\mathit{task}_f\ \texttt{forward}\ (\texttt{async}\ e))]\!] = \\ (\mathit{prm}_f)\ (\texttt{task}\ \texttt{fulfil}(f, \mathcal{C}[\![\texttt{forward}\ (\texttt{async}\ e)]\!]_f))\end{aligned}$$

The optimised compilation of $\mathcal{C}[\![\texttt{forward}\ (\texttt{async}\ e)]\!]_f$ is:

$$(\mathit{prm}_f)\ (\texttt{task}\ E[\texttt{Task}(f, (\lambda \texttt{d}'.\texttt{fulfil}(\texttt{d}', \mathcal{C}[\![e]\!]_{\texttt{d}'}); \texttt{stop})); \texttt{stop}])$$

For comparison, the base compilation gives:

$$(\mathit{prm}_f)\ (\texttt{task}\ E[\texttt{Chain}(f, \texttt{Task}(\texttt{Prom}, (\lambda \texttt{d}'.\texttt{fulfil}(\texttt{d}', \mathcal{C}[\![e]\!]_{\texttt{d}'}); \texttt{stop})), \lambda \texttt{d}'.\lambda x.\texttt{fulfil}(\texttt{d}', x); \texttt{stop}); \texttt{stop}])$$

Types and typing rules are compiled inductively (Fig. 7). The following lemmas guarantee that the compilation strategy does not produce broken target code and state the correctness of the translation.

4.1 Correctness

The correctness of the translation is proven in a number of steps.

The first step involves converting the reduction rules to a labelled transition system where communication via futures is made explicit. This involves splitting several rules involving multiple primitive configurations on the left-hand side to involve single configurations, and labelling the values going into and out of futures. For example, $(\mathit{task}_f\ v)\ (\mathit{fut}_f) \to (\mathit{fut}_f\ v)$ is replaced by the two rules:

$$(\mathit{task}_f\ v) \xrightarrow{\overline{f \downarrow v}} \epsilon \qquad (\mathit{fut}_f) \xrightarrow{f \downarrow v} (\mathit{fut}_f\ v)$$

The other rules introduced are:

$$(\mathit{fut}_f\ v) \xrightarrow{f \uparrow v} (\mathit{fut}_f\ v) \qquad (\mathit{task}_f\ E[\texttt{get}\ h]) \xrightarrow{\overline{h \uparrow v}} (\mathit{task}_f\ E[v])$$

$$(\mathit{chain}_g\ f\ e) \xrightarrow{\overline{f \uparrow v}} (\mathit{task}_g\ e[v/x])$$

Label $f \downarrow v$ captures a value being written to a future, and label $f \uparrow v$ captures a value being read from a future, both from the future's perspective. Labels $\overline{f \downarrow v}$ and $\overline{f \uparrow v}$ are the duals from the perspective of the remainder of the configuration. The remainder of the rules are labelled with τ to indicate that no observable behaviour occurs. The same pattern is applied to the target language.

It is important to note that the values in the labels of the source language are the compiled values, while the values in the labels of the target language remain the same.[6] This is needed so that labelled values such as lambda abstraction match during the bisimulation game.

The composition rules are adapted to propagate or match labels in the standard way. For instance, the rule for matching labels in parallel configurations is:

$$\frac{\mathit{config} \xrightarrow{l} \mathit{config}'' \quad \mathit{config}' \xrightarrow{\bar{l}} \mathit{config}'''}{\mathit{config}\ \mathit{config}' \xrightarrow{\tau} \mathit{config}''\ \mathit{config}'''}$$

The following theorems capture correctness of the translation.

Theorem 1. *If* $\Gamma \vdash \mathit{config}\ \ \mathit{ok}$*, then* $\mathcal{C}[\![\Gamma]\!] \vdash \mathcal{T}[\![\mathit{config}]\!]\ \ \mathit{ok}$*.*

Theorem 2. *If* $\Gamma \vdash \mathit{config}\ \ \mathit{ok}$*, then* $\mathit{config} \sim \mathcal{T}[\![\mathit{config}]\!]$*.*

The first theorem states that translating well-typed configurations results in well-typed configurations. The second theorem states that any well-typed configuration in the source language is bisimilar to its translation. The precise notion of bisimilarity used is bisimilarity up-to expansion [11]. This notion of bisimilarity compresses the administrative, unobservable transitions introduced by the translation.

The proof involves taking each derivation rule in the adapted semantics for the source calculus (described above) and showing that each source configuration is bisimilar to its translation. This is straightforward for the base cases, because tasks are deterministic in both source and target languages, and at most two unobservable transitions are introduced by the translation. To handle the parallel composition of configurations, bisimulation is shown to be compositional, meaning that if $\mathit{config} \sim \mathcal{T}[\![\mathit{config}]\!]$ and $\mathit{config}' \sim \mathcal{T}[\![\mathit{config}']\!]$, then $\mathit{config}\ \ \mathit{config}' \sim \mathcal{T}[\![\mathit{config}\ \ \mathit{config}']\!]$; now by definition $\mathcal{T}[\![\mathit{config}\ \ \mathit{config}']\!] = \mathcal{T}[\![\mathit{config}]\!]\ \mathcal{T}[\![\mathit{config}']\!]$, hence $\mathit{config}\ \ \mathit{config}' \sim \mathcal{T}[\![\mathit{config}]\!]\ \mathcal{T}[\![\mathit{config}']\!]$.

5 Experiments

We benchmarked the implementation of `forward` by comparing it against the blocking pattern `get`-and-`return` and an implementation that uses the `await`-and-`get` (both described in Section 1). The micro-benchmark used is a variant of the broker pattern with 4 workers, compiled with aggresive optimisations (`-O3`). We report the average time (wall clock) and memory consumption of 5 runs of this micro-benchmark under different workloads (Fig. 8). The processing of each message sent involves complex nested loops with quadratic complexity (in the Workload value) written in such a way to avoid the compiler optimising them away — the higher the workload, the higher the probability that the `Broker` actor blocks or awaits in the non-`forward` implementations.

[6] We have omitted the notation from the translation to keep it simple to read

Performance (in seconds)			
Workload	Get	Await+Get	Forward
100	0.03	0.03	0.00
500	0.47	0.25	0.02
1000	1.85	0.94	0.06
3000	16.55	8.29	0.39
5000	45.77	23.01	1.03
7500	103.43	51.62	2.26
10000	183.04	91.86	4.02

Memory consumption (in kilobytes)			
Workload	Get	Await+Get	Forward
100	12697	49446	7334
500	12292	49676	6608
1000	12451	49927	6832
3000	12222	49070	7793
5000	12427	48584	7269
7500	12337	48016	7853
10000	12484	48316	8475

Fig. 8: Elapsed time (left) and memory consumed (right) by the Broker microbenchmark (the lower the better).

The performance results (Fig. 8) show that the `forward` version is always faster than the `get`-and-`return` and `await`-and-`get` version. In the first case, this is expected as blocking prevents the `Broker` actor from processing messages, while the `forward` version does not block. In the second case, we also expected the `forward` version to be faster than the `await`-and-`get`: this is due to the overhead of the context switching operation performed on each `await` statement.

The `forward` version consumes the least amount of memory, while the `await`-and-`get` version consumes the most (Fig. 8). This is expected: `forward` creates one fewer future per message sent than the other two versions; the `await`-and-`get` version has around 5 times more overhead than the `forward` implementation, as it needs to save the context (stack) whenever a future cannot immediately be fulfilled.

Threats to validity The experiments use a microbenchmark, which provides useful information but is not as comprehensive as a case study would be.

6 Related work

Baker discovered futures in 1977 [12]; later Liskov introduced promises to Argus [4]. Around the same time, Halstead introduced implicit futures in Multilisp [13]. Implicit futures do not appear as a first-class construct in the programming language at either the term or type level, as they do in our work.

The `forward` construct was introduced in earlier work [7], in the formalisation of an extension to the active object-based language Creol [14]. The main differences with our work are: our core calculus is much smaller, based on tasks rather than active objects; our calculus includes closures, which complicate the type system, and future chaining; we defined a compilation strategy for `forward`, and benchmark its implementation.

Caromel *et al.* [15] formalise an active object language that transparently handles futures, prove determinism of the language using concepts similar to weak bisimulation, and provide an implementation [16]. In contrast, our work uses a task-based formalism built on top of the lambda calculus and uses fu-

tures explictly. It is not clear whether `forward` can be used in conjunction with transparent futures.

Proving semantics preservation of whole programs is not a new idea [17–22]. We highlight the work from Lochbihler, who added a new phase to the verified, machine-checked Jinja compiler [23] that proves that the translation from multi-threaded Java programs to Java bytecode is semantics preserving, using a delay bisimulation. In contrast, our work uses an on-paper proof using weak bisimilarity up-to expansion, proving that the compilation strategy preserves the semantics of the high-level language.

Ábrahám et al [5] present an extension of the Creol language with promises. The type system uses linear types to track the use of the write capability (fulfilment) of promises to ensure that they are fulfilled precisely once. In contrast to the present work, their type system is significantly more complex, and no `forward` operation is present. Curiously, Encore supports linear types, though lacks promises and hence does not use linear types to keep promises under control.

Niehren et al [6] present a lambda calculus extended with futures (which are really promises). Their calculus explores the expressiveness of programming with promises, by using them to express channels, semaphores, and ports. They also present a linear type system that ensures that promises are assigned only once.

7 Conclusion

One key difference between futures, futures with forward and promises is that the responsibility to fulfil a future cannot be delegated. The `forward` construct allows such delegation, although only of the implicit future receiving the result of some method call, while promises allow arbitrary delegation of responsibility. This paper presented a formal calculus capturing the `forward` construct, which retains the static fulfilment guarantees of futures. A translation of the source calculus into a target calculus based on promises was provided and proven to be semantics preserving. This translation models how `forward` is implemented in the Encore compiler. Microbenchmarks demonstrated that `forward` improves performance in terms of speed and memory overhead compared to two alternative implementations in the Encore language.

References

1. Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017.
2. Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal*

Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.

3. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010.
4. Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 260–267. ACM, 1988.
5. Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *J. Log. Algebr. Program.*, 78(7):491–518, 2009.
6. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.
7. Dave Clarke, Einar Broch Johnsen, and Olaf Owe. Concurrent objects à la carte. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.
8. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
9. Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. ParT: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch-Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016.
10. Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 553–570. ACM, 2013.
11. Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012.
12. Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. *SIGART Newsletter*, 64:55–59, 1977.
13. Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
14. Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006.

15. Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.
16. Denis Caromel, Christian Delbe, Alexandre Di Costanzo, and Mario Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12:issue 1, 2006.
17. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
18. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54. ACM, 2006.
19. Andreas Lochbihler. Verifying a compiler for Java threads. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010.
20. Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65. ACM, 2007.
21. Mitchell Wand. Compiler correctness for parallel languages. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 120–134. ACM, 1995.
22. Xinxin Liu and David Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1995.
23. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.