



**HAL**  
open science

## Godot: All the Benefits of Implicit and Explicit Futures

Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen,  
Tobias Wrigstad

### ► To cite this version:

Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, Tobias Wrigstad. Godot: All the Benefits of Implicit and Explicit Futures. ECOOP 2019 - 33rd European Conference on Object-Oriented Programming, Jul 2019, London, United Kingdom. pp.1-28. hal-02302214

**HAL Id: hal-02302214**


**<https://hal.science/hal-02302214v1>**

Submitted on 1 Oct 2019

**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.

# Godot: All the Benefits of Implicit and Explicit Futures

**Kiko Fernandez-Reyes** 

Uppsala University, Sweden  
kiko.fernandez@it.uu.se

**Dave Clarke** 

Storytel, Stockholm, Sweden

**Ludovic Henrio** 

Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France  
ludovic.henrio@ens-lyon.fr

**Einar Broch Johnsen** 

University of Oslo, Norway  
einarj@ifi.uio.no

**Tobias Wrigstad** 

Uppsala University, Sweden  
tobias.wrigstad@it.uu.se

---

## Abstract

---

Concurrent programs often make use of futures, handles to the results of asynchronous operations. Futures provide means to communicate not yet computed results, and simplify the implementation of operations that synchronise on the result of such asynchronous operations. Futures can be characterised as implicit or explicit, depending on the typing discipline used to type them.

Current future implementations suffer from “future proliferation”, either at the type-level or at run-time. The former adds future type wrappers, which hinders subtype polymorphism and exposes the client to the internal asynchronous communication architecture. The latter increases latency, by traversing nested future structures at run-time. Many languages suffer both kinds.

Previous work offer partial solutions to the future proliferation problems; in this paper we show how these solutions can be integrated in an elegant and coherent way, which is more expressive than either system in isolation. We describe our proposal formally, and state and prove its key properties, in two related calculi, based on the two possible families of future constructs (data-flow futures and control-flow futures). The former relies on static type information to avoid unwanted future creation, and the latter uses an algebraic data type with dynamic checks. We also discuss how to implement our new system efficiently.

**2012 ACM Subject Classification** Software and its engineering → Concurrency control; Software and its engineering → Concurrent programming languages; Software and its engineering → Concurrent programming structures

**Keywords and phrases** Futures, Concurrency, Type Systems, Formal Semantics

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2019.2

**Supplement Material** ECOOP 2019 Artifact Evaluation approved artifact available at <https://dx.doi.org/10.4230/DARTS.5.2.1>

**Funding** Part of this work was funded by the Swedish Research Council, Project 2014-05-545.

## 1 Introduction

Concurrent programs often make use of futures [4] and promises [27], which are handles to possibly not-yet-computed values, that act like a one-off channel for communicating a result from (often a single) producers to consumers. Futures and promises simplify concurrent programming in several ways. Perhaps most importantly, they add elements of structured



© Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad;

licensed under Creative Commons License CC-BY

33rd European Conference on Object-Oriented Programming (ECOOP 2019).

Editor: Alastair F. Donaldson; Article No. 2; pp. 2:1–2:28



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



```

def addition(x: Int, y: Int): Int
  x + y
end
def addition(x: Fut[Int], y: Fut[Int]): Int
  get(x) + get(y)
end

```

■ **Figure 1** *Left*. Data-flow, implicitly typed future, i.e., any argument may be a future value, not visible to the developer. *Right*. Control-flow, explicitly typed future, i.e., the function only accepts future values; synchronisation constructs reduce the future nesting level, e.g., `get`.

programming to message passing, i.e., a message send immediately returns a future, which mimics method calls with a single entry and single exit. This simplifies the control-flow logic, avoids explicit call-backs, and allows a single result to be returned to multiple interested parties – without the knowledge of the producer – through sharing of the future handle. A future is *fulfilled* when a value is associated with it. Futures are further used as synchronisation entities: computations can check if a future is fulfilled (`poll`), block on its fulfilment (`get`), and register a piece of code to be executed on its fulfilment (future chaining – `then`), etc. Promises are similar to (and often blurred with) futures. The main difference is that fulfilment is done manually through a separate first-class handle created at the same time as the future.

Futures are often characterised as either *implicit* or *explicit*, depending on the typing discipline used to type them. Implicit futures are transparent, i.e., it is not generally possible to distinguish in a program’s source whether a variable holds a future value or a concrete value. As a consequence, an operation `x + y` may block if either `x` or `y` are future values. This is called *wait-by-necessity* because blocking operations are hidden from the programmer and only performed when a concrete value is needed. With implicit futures, any function that takes an integer can be used with a future integer, which makes code more flexible and avoids duplication (Fig. 1, *Left*). Explicit futures, in contrast, use future types to distinguish concrete values from future values, e.g., `int` from `Fut[int]`, and rely on an explicit operation, which we will call `get`, to extract the `int` from the `Fut[int]`. The types and the explicit `get` make it clear in the code what operations may cause latency, or block forever. The types also make harder to reuse code that mixes future and concrete values (Fig. 1, *Right*).

Because implicit futures allow future and concrete values to be used interchangeably, they can delay blocking on a future until its value is needed for a computation to move forward. Implementing the same semantics with explicit futures requires considerable effort to deal with any possible combination of future and concrete values at any given juncture.

Programs built from cooperating concurrent processes, like actor programs, commonly compute operations by multiple message sends across several actors, each returning a future. This is implemented by nesting several futures, e.g.,  $f_1 \leftarrow f_2 \leftarrow f_3$  such that  $f_1$  is fulfilled by  $f_2$  which is fulfilled by  $f_3$ . While implicit futures hide these structures by design, explicit futures suffer from a blow-up in the number of `get` operations that must be applied to extract the value, but also in the amount of wrappers that must be added to type the outermost future value. Notably, this makes tail recursive message-passing methods impossible to type as the number of type wrappers must mirror the depth of the recursion.

Futures are important for structuring and synchronising asynchronous activities and have been adopted in mainstream languages like Java [32, 17], C++ [26], Scala [41], and JavaScript [28]. In the actor world, futures reduce complexity considerably by enabling an actor to internally join on the production of several values as part of an operation. Alternative approaches either become visible in an actors interface and require manual “buffering” of intermediate results as part of the actor’s state, or rely on a receive-like construct and the ability to pass an actor any message, which loses the benefit of a clearly defined interface. With the prevalent way of typing futures – as used for example in Java and Scala – a

programmer must choose between introducing blocking to remove future type wrappers [20], or break away from typical structured programming idioms when translating a program into an equivalent actor-based program.

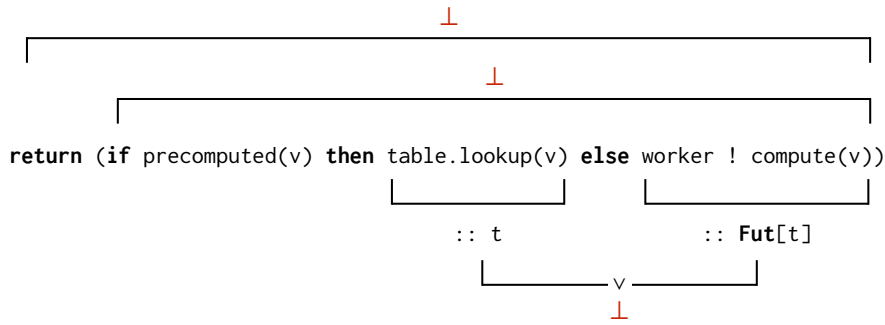
This paper unifies and extends two recent bodies of work on explicit and implicit futures. Henrio [20] observed that the literature directly ties *data flow-driven* and *control flow-driven* futures to the implicit- and explicit-future dichotomy, respectively (e.g., Fig. 1). This work explored the design space of data-flow/control-flow and implicit/explicit dichotomy to support this argument, and developed a combination of data-flow and explicit futures, with a future type that abstracts nesting, avoids the aforementioned explosion of future wrappers and `get` calls for tail recursive methods or pipelines with multiple asynchronous stages, making the chains of futures completely transparent. Fernandez-Reyes et al. [13] proposed an *explicit delegation operation* precisely for handling long (possibly tail recursive) pipelines. Instead of introducing a new future type that hides nesting, this work identifies common delegation patterns in actor-based programs, and proposes a delegation operation that avoids creating unwarranted nested futures. In this system the programmer can control exactly on which stages in a pipeline should be possible to synchronise, reducing the number of created futures.

We distinguish two kinds of futures. We call *control-flow futures* the future constructs that can be implemented by a parametric future type and where each synchronisation blocks until exactly one asynchronous task finishes, the fact that a single fulfilment instruction resolves the blocked state explains the control-flow name. We call *data-flow futures* the future constructs where the synchronisation consists in blocking until a concrete usable value is available, consequently a single synchronisation might wait for the termination of several asynchronous tasks. Data-flow futures are usually implemented by implicit futures.

**Contributions.** This paper shows how to integrate data-flow futures and control-flow futures, and how to seamlessly combine them. We show how data-flow futures can be implemented using control-flow futures and the converse. Our model provides future delegation, data-flow futures, and control-flow futures at the same time, giving the programmer precise control over future access, as well as automatic elision of unnecessary nested futures. More precisely:

- We overview three inherent problems with both explicit and implicit futures that limit their applicability or performance (Section 2).
- We discuss existing mitigation strategies based on typically available future operations or alternatives (Section 3.1) – as well as recent work on data-flow futures [20] and delegation [13] that aim to address overlapping subsets of these problems – and show that *none addresses all of the problems* (Sections 3.2 and 3.3).
- We propose Godot (Section 4), the first system that seamlessly integrates data-flow futures and control-flow futures in a single explicit system. In addition to *addressing all the problems* in Section 2, the system improves on the data-flow explicit futures of [20] by adding support for parametric polymorphism, and improves on the delegation in [13] by allowing it to be applied automatically for data-flow explicit futures.
- We provide two alternative formalisations of Godot (with a common foundation introduced in Section 4.1). FlowFut shows how to extend a data-flow future language with control-flow futures; it is mostly aimed at languages with no current future support (Section 4.2). FutFlow shows how to extend a control-flow future language with data-flow futures; it is aimed at languages with typical explicit future support (Section 4.3).
- We prove progress and type preservation of FlowFut and FutFlow; and
- We introduce a type-driven optimisation strategy for eliding the creation of nested futures (Section 5) and a discussion on the implementation of our system.

In addition to the above, Section 6 discusses related work and Section 7 concludes.



■ **Figure 2** Type Proliferation making code untypable;  $\perp$  denotes the absence of a type for a term.

## 2 Problems Inherent in Explicit and Implicit Futures

Both implicit and explicit futures have limitations. In this section, we overview the problems that exist with existing futures. We use examples presented in pseudocode, where  $o ! m$  and  $o.m$  denote an asynchronous and a synchronous call to a method  $m$  of an object  $o$ , respectively.

**The Type Proliferation Problem.** The way explicit futures are generally added to languages, they end up mirroring the communication structure of a program: the result of an asynchronous operation is typed  $\mathbf{Fut}[t]$ , the result of an asynchronous operation that returns the result of another asynchronous operation is  $\mathbf{Fut}[\mathbf{Fut}[t]]$ , etc. This breaks abstraction and makes code inflexible. For example, consider the following code example that returns values from two different sources. If the answer is precomputed, it is fetched from a table, otherwise the computation is delegated to some worker (see Figure 2 for details).

```
return if precomputed(v) then table.lookup(v) else worker ! compute(v)
```

As denoted by the  $\perp$  types, this is not well-typed as the branches have different types, without any join: `table.lookup(v)` returns a value of type  $t$ , whereas `worker ! compute(v)` returns a  $\mathbf{Fut}[t]$ . Thus, such a common pattern will not work straightforwardly in a program. For similar reasons, tail recursive asynchronous methods are not possible to type as the depth of the recursion must be mirrored in the returned future type. Last, also an effect of the same root cause, explicit futures complicate code reuse – forcing code duplication for operations that should be possible to apply to values of both future and concrete type.

This problem has been previously identified in [16, 20], where the authors showed that there was no direct encoding from implicit futures to explicit futures because an unbounded number of control-flow synchronisations and an unbounded parametric type may be needed to encode a single data-flow future. This is typically the case if one tries to write an asynchronous tail recursive function. For this reason there is no simple encoding of data-flow futures with control-flow futures; Section 4.3 will show how, with a boxing operator and a few changes in the type system, we are able to encode data-flow futures using control-flow futures and to overcome the type resolution problem.

We call this problem, which applies to explicit futures, the *Type Proliferation Problem*.

**The Future Proliferation Problem.** Implicit futures avoid the Type Proliferation Problem by abstracting whether a variable has been computed or not. However, the way implicit futures are generally added to languages, a similar problem appears at run-time. While

tail recursion is possible, running tail calls in constant space is not possible because each recursive call gives rise to an additional future indirection.

The creation of nested futures  $f_1 \leftarrow f_2 \leftarrow f_3$  (etc.) introduces additional latency because the fulfilment of a nest of futures of depth  $n$  adds  $n$  additional operations, which in worst-case must be scheduled separately. Moreover, because a future can be fulfilled with an unfulfilled future, in some implementations, an actor may be falsely deemed schedulable, only to take a step to block on the unfulfilled nested future. For example,  $f_1$  will be “falsely fulfilled” by the unfulfilled future  $f_2$ ; if the activity blocking on  $f_1$  is scheduled to run before  $f_2$  and  $f_3$  are fulfilled, the operation will block again on  $f_2$  or  $f_3$  (possibly both).

This problem, which applies to both implicit and explicit futures, was pointed out in [13]. We call it the *Future Proliferation Problem*.

**The Fulfilment Observation Problem.** The abstraction of implicit futures further loses precision. Consider the following code snippet that could be part of a simple load balancer, that farms out jobs to idle workers, and a call to the load balancer to perform some work.

```
def perform(job : Job) { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job)
```

A call to `perform()` results in a nested future: the outermost future captures whether the load balancer has found an idle worker and successfully delegated the job; the innermost future captures the result of `do_work()`. With explicit futures we can observe the state of the task:

```
get(f)    --block until do_work has been called
get(get(f)) --block until do_work has finished
```

However, with implicit futures, it is not possible to make this distinction as any access will block until the innermost value is returned. Thus, we cannot observe the current stage of such an operation using futures. Concurrent and scheduling library developers need to access the intermediate steps of computations, and this issue hinders the code that they can write.

Similarly, if an unfulfilled future is stored somewhere, say in a hash table implemented by an actor, retrieving it is tricky without accidentally blocking on the production of the future – an unknown operation – rather than the result of `hash_table.lookup()`. Since a hash table may store both concrete and future values due to the nature of implicit futures, knowing when to *not* call `get` on the result of a hash table lookup is not discernible by local reasoning.

This has been highlighted in [21, 20] as the major source of difference between existing implicit and explicit futures. Because of this different behaviour, there is no simple encoding of control-flow futures with data-flow futures. In Section 4.2 we will show such an encoding that relies on a slight adaptation of the type system, and a boxing operator.

This problem applies to implicit futures, we call it the *Fulfilment Observation Problem*.

Following this problem overview, the next section presents existing partial solutions.

### 3 Current Solutions to Future Problems

This section surveys how existing techniques can be used to partially overcome the problems outlined in Section 2. In particular, in Sections 3.2–3.3, we give an informal overview of prior work that this paper amalgamates to address all of the problems in a coherent way.

### 3.1 Standard Mitigation Strategies and Problem Avoidance

**Manual Unpacking of Futures.** Avoiding the Type Proliferation Problem is possible by manually unpacking and returning the concrete value of each future using the aforementioned `get` operation. In the case of the guarded return example, we could write the following:

```
return if precomputed(v) then table.lookup(v) else get(worker ! compute(v))
```

This causes the `else` branch to block its execution until the `compute()` method has finished and is notified of the fulfilment of the current future. This has several problems:

- **Bottleneck.** The enclosing actor is blocked from processing other requests while waiting for `worker ! compute(v)` to finish. This causes subsequent messages to block, even if they could be served from precomputed data. Thus, the blocking `get` introduces a bottleneck.
- **False Fulfilment.** Delaying the return until the concrete value is produced avoids false fulfilment but instead adds an additional step to the operation which adds an unnecessary latency. The task of unpacking the innermost future and fulfilling the outermost must now be scheduled before the client of the outermost future is unblocked. Notably, this changes fulfilment from pull – clients blocking until the value is available, to push – propagating fulfilment of a nested future inwards out. (We revisit this in Section 5.)

Some actor languages that use futures provide a cooperative scheduling construct “`await`” that allows the current method to be suspended pending the fulfilment of a future *without blocking the currently executing actor*. This avoids the bottleneck problem above, but at the same time introduces race conditions due to the possible interleaving of suspended methods – these race conditions only appear through side effects [8].

**Explicit Spawning of a Task.** The explicit creation of a task can be used to solve the Type Proliferation Problem. In the case of the example, the `then` branch spawns a task for something that needed not be asynchronous:

```
return if precomputed(v) then async(table.lookup(v)) else worker ! compute(v)
```

This causes the type checker to accept the program at the expense of performance. The creation of a task involves memory allocation, scheduling of the task, and computation of the task body, which is a simple asynchronous operation. This is feasible, but not optimal.

**Future Chaining to Avoid Blocking and Nesting.** Future chaining can be used to avoid unnecessary blocking in some cases. Future chaining supports the construction of pipelines of futures which are not nested, but still need to be represented at run-time. For example, here is how we could add the result of `worker ! compute(v)` to the table of precomputed values (so it effectively becomes a cache) *without* delaying the returning of the result to a client:

```
var result = worker ! compute(v)
result.then(fun r => this.table.add(v, r))
return result
```

The `then` method attaches a callback function that will be run upon the fulfilment of `result`, with `r` bound to the value used to fulfil `result`. Although the callback registration happens before the `return`, the execution of the registered function does not happen until after the future is fulfilled, meaning it causes no delay.

While chaining can avoid some Type Proliferation, it does not enable tail recursive calls.

**Changing the Program Structure: Replace Return with Message Send.** An alternative solution is to give up on structured programming ideals and instead of returning values back up the call stack, instruct the producer of a value how to communicate the result to its consumers. Here is an example of how that might look in the Type Proliferation Problem example:

```
if precomputed(v) then client ! receive(table.lookup(v)) --send result to client
else worker ! compute(v, client) --pass client id to worker
```

With this design, a method that previously returned a value must be passed the identity of the consumer of the result as an argument (possibly a list of consumers) to explicitly send the result to the consumer(s) according to some agreed-upon protocol. Instead of `id(s)`, it can take as input some lambda function that know how to communicate the result back to interested parties. A downside of this solution is that the consumers must be known at the time of the call. This is in contrast to a caller sharing a returned future with whoever might be interested in the result after the call is made.

This solution requires the existence of a specific method in the consumer for each operation and causes an operation to be spread over multiple methods. Submitting multiple jobs for execution requires manually handling the possibility of the results coming back in any order, and possibly provide multiple different methods for getting the results.

Returning values differently from synchronous and asynchronous computations increases complexity for functions and data structures that should be usable in both contexts. This is typical in, e.g., Cilk [6] where a function can be “spawned” asynchronously or called synchronously, and in many actor languages (e.g., Joelle [10], ABS [22] and Encore [7]) where an actor’s interface is asynchronous externally but synchronous internally.

**Changing the Program Structure: Use Promises Instead of Futures.** Both the Type Proliferation Problem and the Future Proliferation Problem can be overcome by resorting to manually handled *promises*: instead of passing the identity of the recipient around, we pass around a pointer to a shared space where the result can be stored. Promises are similar to futures, but are less transparent and, because they are manipulated explicitly both on the side of the producer and the consumer, lack many of the guarantees of futures: promises are created and fulfilled manually and are thus not guaranteed to be fulfilled at all, may be fulfilled more than once, possibly by several actors.<sup>1</sup> With this design, workers are passed a promise created by a client. Upon finishing the work, the worker fulfils the promise.

### 3.2 Data-flow Explicit Futures

Henrio [20] observed that the traditional dichotomy of implicit and explicit futures was focusing mainly on typing and not on how futures are synchronised, and proposed an alternative categorisation: *control-flow* futures and *data-flow* futures, depending on how the synchronisation on futures works. With control-flow synchronisation, each nested future must be explicitly unpacked using `get` to return another future or a concrete value. Data-flow synchronisation is wait-by-necessity as usual for implicit futures: nesting is invisible, and a `get` always returns a concrete value, even from a nested future. Separating typing from synchronisation allows new combinations of future semantics, such as explicit data-flow futures, which address the Type Proliferation Problem of Section 2.

<sup>1</sup> Futures have static fulfilment guarantees, they are implicitly fulfilled, unless the fulfilling computation gets stuck. Promises have no static fulfilment guarantees, even when the program is not stuck.



The traditional way of typing explicit futures, by a parametric type, has always led to control-flow synchronisation on futures while data-flow futures had no future type. Data-flow synchronisation naturally leads to an alternative type system called *DeF*, such that the run-time structure of futures is no longer mirrored by their type. Instead, a **Fut**[ $\tau$ ] type represents *zero* or more nested futures – the *zero* means that a concrete value may appear as a future value. This allows future-typed code to be reused with concrete values but also allows tail recursion and methods returning either a concrete value or a future. In the Type Proliferation Problem, the branches would still have different types ( $\tau$  and **Fut**[ $\tau$ ]), but  $\tau$  can be lifted to **Fut**[ $\tau$ ], collapsing the **Fut**[**Fut**[ $\tau$ ]] returned by the entire asynchronous expression into a **Fut**[ $\tau$ ]. Let the keyword **async** denote the spawning of an asynchronous task.

```
async (if precomputed(v) then table.lookup(v) else worker ! compute(v))
```

*Data-flow explicit futures address the Type Proliferation Problem but it does not address the Future Proliferation Problem or the Fulfilment Observation Problem.*

**A Formal Introduction to DeF.** For simplicity and to align with upcoming sections, we adapt Henrio’s DeF calculus to a concurrent, lambda-based calculus. We use an **async** construct to spawn tasks and a **get** construct for data-flow synchronisation on a future. The types are the basic types  $\mathcal{K}$ , abstraction and futures.

$$\begin{array}{l} \text{Expressions } e ::= v \mid e e \mid \mathbf{return} e \mid \mathbf{async} e \mid \mathbf{get} e \\ \text{Values } v ::= c \mid x \mid f \mid \lambda x.e \\ \text{Types } \tau ::= \mathcal{K} \mid \tau \rightarrow \tau \mid \mathbf{Fut} \tau \\ \text{Evaluation context } E ::= \bullet \mid E e \mid v E \mid \mathbf{return} E \mid \mathbf{get} E \end{array}$$

The operational semantics use a small-step reduction semantics with reduction-based, contextual rules for evaluation within tasks. An evaluation context  $E$  contains a hole  $\bullet$  that denotes where the next reduction step happens. Configurations consist of tasks ( $task_f e$ ), unfulfilled futures ( $fut_f$ ) and fulfilled futures ( $fut_f v$ ). When a task finishes, i.e., reduces to a value  $v$ , the corresponding future is fulfilled with  $v$ .

We show the most interesting reduction rules in Figure 3: RED-ASYNC spawns a new computation and puts a fresh future in place of the spawned expression. RED-GET-VAL applies **get** to a concrete value which reduces to the value itself. RED-GET-FUT applies **get** on a future chain of length  $\geq 1$ , reducing it future by future. A run-time test,  $isfut?(v)$ , is required to check whether  $v$  is a future value or a concrete value.

Figure 3 shows the most interesting type rules. We first have two sub-typing rules: a concrete value can be typed as a future, and nested future types are unnecessary. By T-ASYNC, any well-typed expression of type  $\tau$  can be spawned off in an asynchronous task that returns a **Fut**  $\tau$ . By T-GET, **get** can be applied to unpack a **Fut**  $\tau$ , yielding a value of type  $\tau$ .

**Summary.** Data-flow futures allow the programmer to focus on expressing future-like algorithms without explicitly manipulating every synchronisation point. A single future and multiple nested futures are indistinguishable with respect to types and synchronisation. Because the type system allows the implicit lifting of a concrete value to a (fulfilled) future value, code that uses futures can be reused with concrete values.

Reduction rules:  $e \rightarrow e'$

$$\frac{\text{(RED-ASYNC)} \quad \text{fresh } f}{(task_g E[\mathbf{async} e]) \rightarrow (fut_f) (task_f e) (task_g E[f])}$$

$$\frac{\text{(RED-GET-VAL)} \quad \neg isfut?(v)}{(task_f E[\mathbf{get} v]) \rightarrow (task_f E[v])}$$

$$\frac{\text{(RED-GET-FUT)} \quad isfut?(g)}{(task_f E[\mathbf{get} g]) (fut_g v) \rightarrow (task_f E[\mathbf{get} v]) (fut_g v)}$$

Subtyping:

$$\tau <: \mathbf{Fut} \tau$$

$$\mathbf{Fut} (\mathbf{Fut} \tau) <: \mathbf{Fut} \tau$$

Typing rules:  $\Gamma \vdash_\rho e : \tau$

$$\text{(T-ASYNC)}$$

$$\Gamma \vdash_\tau e : \tau$$

$$\Gamma \vdash_\rho \mathbf{async} e : \mathbf{Fut} \tau$$

$$\text{(T-GET)}$$

$$\Gamma \vdash_\rho e : \mathbf{Fut} \tau$$

$$\Gamma \vdash_\rho \mathbf{get} e : \tau$$

■ **Figure 3** Reduction and typing rules for data-flow explicit futures.

### 3.3 Delegating Future Fulfilment

To avoid the Type Proliferation Problem and Future Proliferation Problem of Section 2, Fernandez-Reyes et al. [13] proposed a delegation construct that delegates the fulfilment of the current-in-call future to another task in the context of control-flow explicit futures. This **forward** construct supports tail-recursive asynchronous methods and allows them to run in *constant space*, because only a single future is needed.<sup>2</sup> The Fulfilment Observation Problem is avoided because of the control-flow synchronisation. Library code can distinguish the futures it manipulates and the concrete values that client programs are interested in.

In contrast to DeF, delegation requires an explicit keyword. This can be seen in the Type Proliferation Problem example by inserting **return** in the **then**-branch and **forward** in the **else**-branch. In the **then**-branch, the concrete value is returned; in the **else**-branch, **forward** delegates to a worker to *fulfil the current future*. In both cases, the return type is **Fut**[t]. This shows how a method's return type no longer needs to (but may) mirror the internal communication structure of a method in order to avoid the Fulfilment Observation Problem:

```
async (if precomputed(v) then return table.lookup(v)
      else forward worker ! compute(v))
```

*Delegation and explicit future types address the Future Proliferation Problem and Fulfilment Observation Problem, but only in part the Type Proliferation Problem – reuse is still limited by future types, causing code duplication or blocking to remove future types.*

**A Formal Introduction to Forward.** We present the semantics of delegation similarly through a concurrent, lambda-based calculus, adapted from Fernandez-Reyes' work. The syntax reuses the concepts from the previous section and adds the **forward** construct which transfers the obligation to fulfil a future to another task and future chaining (**then**( $e, e$ )), which registers a piece of code to be executed on its fulfilment. While the latter is not strictly necessary, its run-time semantics are necessary to express the semantics of **forward**, so explicit support for future chaining adds very little complexity. The types are the same as in the previous calculus except that there is no subtyping rule. The typing judgement has an extra parameter,  $\rho$ , which prevents the use of **forward** under certain circumstances (explained later).

<sup>2</sup> This cannot be observed in Fig. 3 because we have omitted the compilation optimisations [13]. This optimisations follow the same logic as Section 5.

$$e ::= \dots \mid \mathbf{then}(e, e) \mid \mathbf{forward} e \quad E ::= \dots \mid \mathbf{then}(E, e) \mid \mathbf{then}(v, E) \mid \mathbf{forward} E$$

We show the most interesting reduction rules in Figure 4: RED-GET captures blocking synchronisation through **get** on a future  $f$ . RED-CHAIN-NEW attaches a callback  $e$  on a future  $f$  to be executed (rule RED-CHAIN-RUN) once  $f$  is fulfilled. Chaining on a future immediately returns another future which will be fulfilled with the result of the callback. RED-FORWARD captures delegation. Like **return** it immediately finishes the current task, replacing it with a “chain task” that will fulfil the same future as the removed task. This **chain** will be executed when the delegated task is finished, i.e., when the future  $h$  is fulfilled.

Reduction rules:  $e \rightarrow e'$

$$\begin{array}{c}
 \text{(RED-GET)} \\
 (task_f E[\mathbf{get} h]) (fut_h v) \rightarrow (task_f E[v]) (fut_h v) \\
 \\
 \text{(RED-CHAIN-RUN)} \\
 (chain_g f e) (fut_f v) \rightarrow (task_g (e v)) (fut_f v) \\
 \\
 \text{(RED-CHAIN-NEW)} \\
 \frac{fresh\ g}{(task_f E[\mathbf{then}(h, e)]) \rightarrow (fut_g) (chain_g h \lambda x.e) (task_f E[g])} \\
 \\
 \text{(RED-FORWARD)} \\
 (task_f E[\mathbf{forward} h]) \rightarrow (chain_f h \lambda x.x)
 \end{array}$$

Typing rules:  $\Gamma \vdash_\rho e : \tau$

$$\begin{array}{c}
 \text{(T-CHAIN)} \\
 \frac{\Gamma \vdash_\rho e : \mathbf{Fut} \tau \quad \Gamma, x : \tau \bullet, e' : \tau'}{\Gamma \vdash_\rho \mathbf{then}(e, e') : \mathbf{Fut} \tau'} \\
 \\
 \text{(T-FORWARD)} \\
 \frac{\Gamma \vdash_\rho e : \mathbf{Fut} \rho \quad \rho \neq \bullet}{\Gamma \vdash_\rho \mathbf{forward} e : \tau}
 \end{array}$$

■ **Figure 4** Reduction and typing rules of forward calculus.

The most interesting type rules deal with future chaining and forward. By T-FORWARD, fulfilment of the current future can be delegated to any expression returning a future. The requirement  $\rho \neq \bullet$  prevents the use of **forward** inside lambda expressions. Otherwise, a lambda could be sent to another task and run in a context different from its defining context, which could inadvertently modify the return type of a task, leading to unsoundness. By T-FORWARD, any type can be used as the result type. Since **forward** halts the execution of the current task, there is no traditional return value from **forward**, which makes this practice sound. T-CHAIN types the chaining on the result of any expression returning a future.

**Summary.** Delegation allows the programmer to push the fulfilment of the *current-in-call future* to another task, thereby avoiding future nesting both in types and at run-time. Here, the result of **get** can be another future and a concrete value cannot be used when a future is expected. While Future Proliferation is avoided, the programmer needs to explicitly insert delegation points and there are restrictions on code reuse with and without future values.

## 4 Godot: Integrating Data- and Control-Flow Futures and Delegation

The core contribution of this paper is Godot [5], a system that seamlessly integrates data-flow explicit futures and control-flow explicit futures, and extends them to increase expressiveness while reducing the number of future values needed at run-time. The resulting system uses **forward**-style *implicitly* on data-flow futures. For clarity, in the sequel, control-flow futures will retain the **Fut**  $\tau$  type, and data-flow futures will be denoted by **Flow**  $\tau$ .

## 4.1 Design Space and Formal Semantics

Godot is formalised as two distinct versions of a core calculus using a concurrent, task-based, modified version of System F: FlowFut that uses data-flow futures as primitives and uses them to encode control-flow futures (Section 4.2); and FutFlow that uses control-flow futures as primitives and uses them to encode data-flow futures (Section 4.3). The target audience for FlowFut is language designers who wish to add Godot to a language without futures. The target audience for FutFlow is language designers who wish to incorporate Godot in a language that already supports control-flow futures.

The core calculus contains tasks, control-flow futures and data-flow futures, and operations on them. For simplicity, we abstract from mutable state, as this would detract from the main points. We use explicit futures, recall that control-flow futures are typed by **Fut**  $\tau$  and data-flow futures by **Flow**  $\tau$ . Operations on data-flow futures are distinguished by a  $\star$ , e.g., **get** operates on **Fut**  $\tau$  and **get\*** operates on **Flow**  $\tau$ , etc.

The calculus consists of two levels: configurations and expressions. Configurations represent the run-time as a collection of concurrent tasks, futures, and asynchronous chained operations. Expressions correspond to programs and what tasks evaluate to. A task represents a unit of work and its result is placed in either a flow or future abstraction, depending on the type system. A task represents any asynchronous computation, it can for example correspond to a runnable task in Java, or a message treatment in actor and active object languages.

Chaining operations on either data-flow and control-flow futures attaches a closure to the future that will be schedulable when the future is fulfilled. Abstracting from mutable state, we cannot model the consequences of closures with side effects, but we can easily integrate any pre-existing approach, e.g., [9]. With respect to the simple calculi in Section 3, we add a **return** expression which immediately finishes a task with a given return value. This expression has been added to show how we reduce the creation of futures upon returning from a task with respect to data-flow futures. The **return** construct shares limitations with the **forward** construct, which we explain in the coming subsections.

The remainder of Section 4.1 introduces parts of the language that are common to both calculi: run-time configurations, types, and their static and run-time semantics. We delay the presentation of expressions and values, their static and run-time semantics and the type and term encodings of one future type in terms of the other to Sections 4.2 and 4.3.

**Syntax.** The calculus contain run-time configurations, expressions, and values.

$$config ::= \epsilon \mid (flow_f) \mid (flow_f v) \mid (fut_f) \mid (fut_f v) \mid (task_f e) \mid (chain_f f e) \mid config\ config$$

Configurations represent running programs. A global configuration *config* represents the global state of the system, e.g.,  $(task_f e)$   $(flow_f)$  represents a global configuration with a single task running expression  $e$ , whose result will fulfil flow  $f$ . Partial configurations *config* show a view of the state of the program, and are multisets of unfulfilled futures  $((flow_f)$  and  $(fut_f)$ ), fulfilled futures  $((flow_f v)$  and  $(fut_f v)$ ), tasks  $(task_f e)$ , and chains  $(chain_f f e)$ , where the empty configuration is  $\epsilon$  and multiset union is denoted by whitespace.

Note that *flow* and *fut* configurations do not co-exist. Depending on the calculus, a task fulfils either a *flow* or a *fut*. This distinction is clarified in each respective calculus.

**Static Semantics.** The types,  $\tau ::= \mathcal{K} \mid \tau \rightarrow \tau \mid X \mid \forall X.\tau \mid \mathbf{Flow}\tau \mid \mathbf{Fut}\tau$ , are the common basic types ( $\mathcal{K}$ ), abstraction ( $\tau \rightarrow \tau$ ), type variables ( $X$ ), universal quantification ( $\forall X.\tau$ ), flow types (**Flow**  $\tau$ ) and future types (**Fut**  $\tau$ ). In the typing rules, we assume that

$$\begin{array}{c}
\text{(T-UFLOW)} \\
\frac{f \in \text{dom}(\Gamma)}{\Gamma \vdash (\text{flow}_f) \text{ ok}} \\
\\
\text{(T-TASKFLOW)} \\
\frac{f : \mathbf{Flow} \tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (\text{task}_f e) \text{ ok}} \\
\\
\text{(T-FFLOW)} \\
\frac{f : \mathbf{Flow} \tau \in \Gamma \quad \Gamma \vdash_\bullet v : \tau}{\Gamma \vdash (\text{flow}_f v) \text{ ok}} \\
\\
\text{(T-UFUT)} \\
\frac{f \in \text{dom}(\Gamma)}{\Gamma \vdash (\text{fut}_f) \text{ ok}} \\
\\
\text{(T-TASKFUT)} \\
\frac{f : \mathbf{Fut} \tau \in \Gamma \quad \Gamma \vdash_\tau e : \tau}{\Gamma \vdash (\text{task}_f e) \text{ ok}} \\
\\
\text{(T-FFUT)} \\
\frac{f : \mathbf{Fut} \tau \in \Gamma \quad \Gamma \vdash_\bullet v : \tau}{\Gamma \vdash (\text{fut}_f v) \text{ ok}} \\
\\
\text{(T-CHAINFLOW)} \\
\frac{f : \mathbf{Flow} \tau \in \Gamma \quad g : \mathbf{Flow} \tau' \in \Gamma \quad \Gamma \vdash_\tau e : \tau' \rightarrow \tau}{\Gamma \vdash (\text{chain}_f g e) \text{ ok}} \\
\\
\text{(T-CHAINFUT)} \\
\frac{f : \mathbf{Fut} \tau \in \Gamma \quad g : \mathbf{Fut} \tau' \in \Gamma \quad \Gamma \vdash_\tau e : \tau' \rightarrow \tau}{\Gamma \vdash (\text{chain}_f g e) \text{ ok}} \\
\\
\text{(T-EMPTY)} \\
\frac{}{\Gamma \vdash \diamond \text{ ok}} \\
\\
\text{(T-CONFIG)} \\
\frac{\Gamma \vdash \text{config}_1 \text{ ok} \quad \text{defs}(\text{config}_1) \cap \text{defs}(\text{config}_2) = \emptyset \quad \Gamma \vdash \text{config}_2 \text{ ok} \quad \text{writers}(\text{config}_1) \cap \text{writers}(\text{config}_2) = \emptyset}{\Gamma \vdash \text{config}_1 \text{ config}_2 \text{ ok}} \\
\\
\text{(T-GCONFIG)} \\
\frac{\Gamma \vdash \text{config} \text{ ok} \quad \text{dom}(\Gamma) = \text{defs}(\text{config})}{\Gamma \vdash \text{config}}
\end{array}$$

■ **Figure 5** Well-formed configurations. The helper functions  $\text{defs}(\text{config})$  and  $\text{writers}(\text{config})$  extract the set of futures (data-flow and control-flow) or writers of futures in a configuration.

the types of the premises are *normalised*. We denote the normalised type  $\tau$  by  $\downarrow\tau$ , i.e., the type  $\tau$  with flattened flow types, defined inductively:

$$\begin{array}{l}
\downarrow K = K \quad \downarrow X = X \quad \downarrow \forall X. \tau = \forall \downarrow X. \downarrow \tau \quad \downarrow (\tau \rightarrow \tau') = \downarrow \tau \rightarrow \downarrow \tau' \\
\downarrow \mathbf{Flow} (\mathbf{Flow} \tau) = \downarrow \mathbf{Flow} \tau \quad \downarrow \mathbf{Flow} \tau = \mathbf{Flow} \downarrow \tau \text{ if } \tau \neq \mathbf{Flow} \tau' \quad \downarrow \mathbf{Fut} \tau = \mathbf{Fut} \downarrow \tau
\end{array}$$

**Well-Formed Configurations.** Type judgements  $\Gamma \vdash \text{config} \text{ ok}$  express that configurations are well-formed in an environment  $\Gamma$  that gives the types of futures (Figure 5). Unfulfilled flow and future configurations are well-formed if their variable  $f$  exists in the environment (T-UFLOW, T-UFUT). Tasks are well-formed if their body is well-typed with the type of the future or flow they are fulfilling (T-TASKFLOW, T-TASKFUT).

The meaning of  $\Gamma \vdash_\rho e : \tau$  is that  $e$  has type  $\tau$  under  $\Gamma$  inside a task whose static return type is  $\rho$ , where  $\rho ::= \tau \mid \bullet$ . Once the concrete syntax is introduced for the two calculi, this notation is used to express that a **return** inside  $e$  must return a value of type  $\rho$ . The special form  $\bullet$  of  $\rho$  disallows the use of **return**. Thus, by (T-FFLOW) and (T-FFUT), values of fulfilled flow configurations cannot be lambda expressions containing a **return** expression. Chained configurations are well-formed if their bodies are well-typed. Note that the body must be a lambda function (T-CHAINFLOW, T-CHAINFUT).

Configurations are well-formed if all sub configurations have disjoint futures and there are not two tasks writing to the same future (T-CONFIG, T-GCONFIG). (The definitions of auxiliary functions  $\text{defs}()$  and  $\text{writers}()$  are straightforward.) These side conditions ensure that there are no races on fulfilment.

**Dynamic Semantics.** Configurations consist of a multiset of tasks, data-flow futures and chained configurations with an initial program configuration ( $\text{flow}_{f_{\text{main}}}$ ) ( $\text{task}_{f_{\text{main}}} e$ ), where  $f_{\text{main}}$  is fulfilled by the result of  $e$  at the end of execution. Configurations are commutative monoids under configuration concatenation, with  $\epsilon$  as unit (Figure 6). The configuration evaluation rules (Figure 6) 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.

*Equivalence relation*

$$\text{config } \epsilon \equiv \epsilon \text{ config} \quad \text{config config}' \equiv \text{config}' \text{ config} \quad \text{config} (\text{config}' \text{ config}'') \equiv (\text{config config}') \text{ config}''$$

*Configuration run-time*

$$\begin{array}{c} \text{(R-FULFILFLOWVALUE)} \\ \frac{\neg \text{isflow?}(v)}{(\text{task}_f v) (\text{flow}_f) \rightarrow (\text{flow}_f v)} \\ \\ \text{(R-FUTFULFILVALUE)} \\ \frac{v \neq \blacklozenge v'}{(\text{task}_f v) (\text{fut}_f) \rightarrow (\text{fut}_f v)} \\ \\ \text{(R-CHAINRUNFLOW)} \\ (\text{chain}_g f e) (\text{flow}_f v) \rightarrow (\text{task}_g (e v)) (\text{flow}_f v) \\ \\ \text{(R-CONFIG)} \\ \frac{\text{config} \rightarrow \text{config}''}{\text{config config}' \rightarrow \text{config}'' \text{ config}'} \end{array} \quad \begin{array}{c} \text{(R-FULFILFLOW)} \\ \frac{\text{isflow?}(g)}{(\text{task}_f g) \rightarrow (\text{chain}_f g \lambda x.x)} \\ \\ \text{(R-FLOWCOMPRESSION)} \\ (\text{task}_f \blacklozenge g) \rightarrow (\text{chain}_f g \lambda x.x) \\ \\ \text{(R-CHAINRUNFUT)} \\ (\text{chain}_g f e) (\text{fut}_f v) \rightarrow (\text{task}_g (e v)) (\text{fut}_f v) \\ \\ \text{(R-CONFIGEQUIV)} \\ \frac{\text{config} \equiv \text{config}' \quad \text{config}' \rightarrow \text{config}'' \quad \text{config}'' \equiv \text{config}'''}{\text{config} \rightarrow \text{config}'''} \end{array}$$

■ **Figure 6** Configuration run-time and configuration equivalence rules modulo associativity and commutativity.  $\blacklozenge v$  represents the encoding of a data-flow future in terms of a control-flow futures.

## 4.2 FlowFut: Primitive Data-Flow and Encoded Control-Flow Futures

This section presents FlowFut which instantiates the expression syntax of Godot presented in the previous section. FlowFut has primitive support for data-flow futures and support for control-flow futures as an extension, using an encoding in terms of data-flow futures. We first describe a sublanguage that only has data-flow futures before extending it with control-flow futures. FlowFut illustrates how to extend a language with data-flow future like ProActive [3], JavaScript, or DeF [20] to support control-flow futures. Note that DeF is the only language that has explicit data-flow futures but it has currently no implementation.

The FlowFut sublanguage contain expressions and values:

$$\begin{array}{l} e ::= v \mid e e \mid e [\tau] \mid \mathbf{return} e \mid \mathbf{async}^* e \mid \mathbf{get}^* e \mid \mathbf{then}^*(e, e) \mid \square e \mid \mathbf{unbox} e \\ v ::= c \mid x \mid f \mid \lambda x.e \mid \lambda X.e \mid \square v \end{array}$$

Expressions are values ( $v$ ), application ( $e e$ ), type application ( $e [\tau]$ ), the return of expressions ( $\mathbf{return} e$ ), spawning an asynchronous task returning a data-flow future ( $\mathbf{async}^* e$ ), blocking on the fulfilment of a data-flow future ( $\mathbf{get}^* e$ ) and future chaining to attach a callback on a future to be executed on the future's fulfilment ( $\mathbf{then}^*(e, e)$ ). To support the encoding of control-flow futures, a lifting operation that we call boxing is introduced ( $\square e$ ) together with a dual unboxing operation ( $\mathbf{unbox} e$ ). Values are constants, variables, data-flow futures, abstraction, and type abstraction. Additionally, a value may be boxed ( $\square v$ ).

**Static Semantics.** The type system has the common types except the control-flow future type ( $\mathbf{Fut} \tau$ ). In its stead, we use a type encoded in terms of data-flow futures,  $\square \tau$ . We show explicit flattening rules for the encodings of control-flow futures in terms of data-flow futures.

$$\begin{array}{l} \text{Types:} \quad \tau ::= \mathcal{K} \mid \tau \rightarrow \tau \mid X \mid \forall X.\tau \mid \mathbf{Flow} \tau \mid \square \tau \\ \text{Previous flattening rules and:} \quad \downarrow \square \tau = \square \downarrow \tau \end{array}$$

$$\begin{array}{c}
\text{(TF-ENV)} \\
\frac{}{\vdash \epsilon} \\
\text{(TF-ENVEXPR)} \\
\frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau} \\
\text{(TF-ENVVAR)} \\
\frac{X \notin \text{dom}(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, X} \\
\text{(TF-K)} \\
\frac{\vdash \Gamma}{\Gamma \vdash \mathcal{K}} \\
\text{(TF-FLOW)} \\
\frac{\Gamma \vdash \tau \quad \tau \neq \mathbf{Flow} \tau'}{\Gamma \vdash \mathbf{Flow} \tau} \\
\text{(TF-ARROW)} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash \tau \rightarrow \tau'} \\
\text{(TF-X)} \\
\frac{X \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash X} \\
\text{(TF-FORALL)} \\
\frac{\Gamma, X \vdash \tau}{\Gamma \vdash \forall X. \tau} \\
\text{(Box)} \\
\frac{\Gamma \vdash \tau}{\Gamma \vdash \square \tau}
\end{array}$$

■ **Figure 7** Type formation rules where  $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, X$ .

$$\begin{array}{c}
\text{(T-CONSTANT)} \\
\frac{c \text{ has type } \mathcal{K} \quad \Gamma \vdash \mathcal{K}}{\Gamma \vdash_{\rho} c : \mathcal{K}} \\
\text{(T-VARIABLE)} \\
\frac{x : \tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_{\rho} x : \tau} \\
\text{(T-FLOW)} \\
\frac{f : \mathbf{Flow} \tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_{\rho} f : \downarrow \mathbf{Flow} \tau} \\
\text{(T-VALFLOW)} \\
\frac{\Gamma \vdash_{\rho} e : \tau}{\Gamma \vdash_{\rho} e : \downarrow \mathbf{Flow} \tau} \\
\text{(T-RETURN)} \\
\frac{\Gamma \vdash_{\tau} e : \tau \quad \tau \neq \bullet \quad \Gamma \vdash \tau'}{\Gamma \vdash_{\tau} \mathbf{return} e : \tau'} \\
\text{(T-ABSTRACTION)} \\
\frac{\Gamma, x : \tau \vdash_{\bullet} e : \tau'}{\Gamma \vdash_{\rho} \lambda x. e : \tau \rightarrow \tau'} \\
\text{(T-BOX)} \\
\frac{\Gamma \vdash_{\rho} e : \tau}{\Gamma \vdash_{\rho} \square e : \square \tau} \\
\text{(T-UNBOX)} \\
\frac{\Gamma \vdash_{\rho} e : \square \tau}{\Gamma \vdash_{\rho} \mathbf{unbox} e : \tau} \\
\text{(T-APPLICATION)} \\
\frac{\Gamma \vdash_{\rho} e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash_{\rho} e_2 : \tau}{\Gamma \vdash_{\rho} e_1 e_2 : \tau'} \\
\text{(T-TYPEABSTRACTION)} \\
\frac{\Gamma, X \vdash_{\bullet} e : \tau}{\Gamma \vdash_{\rho} \lambda X. e : \downarrow \forall X. \tau} \\
\text{(T-TYPEAPPLICATION)} \\
\frac{\Gamma, X \vdash_{\rho} e : \forall X. \tau'}{\Gamma \vdash_{\rho} e [\tau] : \downarrow \tau' [\tau/X]} \\
\text{(T-ASYNCSTAR)} \\
\frac{\Gamma \vdash_{\tau} e : \tau}{\Gamma \vdash_{\rho} \mathbf{async}^* e : \downarrow \mathbf{Flow} \tau} \\
\text{(T-GETSTAR)} \\
\frac{\Gamma \vdash_{\rho} e : \mathbf{Flow} \tau}{\Gamma \vdash_{\rho} \mathbf{get}^* e : \tau} \\
\text{(T-THENSTAR)} \\
\frac{\Gamma \vdash_{\rho} e_1 : \mathbf{Flow} \tau' \quad \Gamma \vdash_{\tau} e_2 : \tau' \rightarrow \tau}{\Gamma \vdash_{\rho} \mathbf{then}^*(e_1, e_2) : \downarrow \mathbf{Flow} \tau}
\end{array}$$

■ **Figure 8** Typing of expressions where futures are encoded as  $\mathbf{Fut} \tau \hat{=} \square \mathbf{Flow} \tau$ .

**Well-Typed Expressions.** The type formation rules are given in Figure 7 and the typing rules are given in Figure 8. In places where a **return** may appear,  $\rho$  is some  $\tau$ , the return type of the task,  $\rho$ , otherwise  $\bullet$ , which makes **return** ill-typed. This (or something equivalent) is necessary – otherwise passing a lambda that contains a **return** to another task might change the return type of the *task*, not of the expression.

The type rules consist of the common System F typing rules: typing of a constant (T-CONSTANT), typing variables (T-VARIABLE), the abstraction typing rule (T-ABSTRACTION) that sets the return type of the task to  $\bullet$ , preventing **return** in lambdas, and application (T-APPLICATION). Type abstraction and application are the common ones with the distinctive flattening of the types (T-TYPEABSTRACTION and T-TYPEAPPLICATION). The rules regarding  $\mathbf{Flow} \tau$  types state that an expression of type  $\tau$  can be lifted to a  $\mathbf{Flow} \tau$  (T-VALFLOW), spawning a task returns a data-flow future type and the spawned task sets its returned type to that of the expression running asynchronously (T-ASYNCSTAR). The constructs **get** $^* e$  returns the content of the data-flow future (T-GETSTAR). Chaining on a data-flow future adds a callback to expression  $e_1$ , returning immediately a new data-flow future (T-THENSTAR). Control-flow futures are encoded in terms of data-flow futures with the  $\square e$  operator with type  $\square \tau$ , where  $\mathbf{Fut} \tau \hat{=} \square \tau$ .

**Dynamic Semantics.** Configurations are as in the previous section, except using control-flow futures. Thus, the initial program configuration is  $(\mathit{fut}_{f_{\text{main}}}) (\mathit{task}_{f_{\text{main}}} e)$ , where  $f_{\text{main}}$  is fulfilled by the result of  $e$  at the end of execution. The dynamic semantics are formulated

$$\begin{array}{c}
\text{(R-}\beta\text{)} \\
\frac{}{(task_f E[\lambda x.e v]) \rightarrow (task_f E[e[v/x]])} \\
\text{(R-TYPEAPPLICATION)} \\
\frac{}{(task_f E[(\lambda X.e) [\tau]]) \rightarrow (task_f E[e[\tau/X]])} \\
\text{(R-GETSTAR)} \\
\frac{isflow?(g)}{(task_f E[\mathbf{get*} g]) (flow_g v) \rightarrow (task_f E[v]) (flow_g v)} \\
\text{(R-GETVAL)} \\
\frac{\neg isflow?(v)}{(task_f E[\mathbf{get*} v]) \rightarrow (task_f E[v])} \\
\text{(R-ASYNCSTAR)} \\
\frac{fresh f}{(task_g E[\mathbf{async*} e]) \rightarrow (flow_f) (task_f e) (task_g E[f])} \\
\text{(R-RETURN)} \\
\frac{}{(task_f E[\mathbf{return} v]) \rightarrow (task_f v)} \\
\text{(R-CHAINRUNFLOW)} \\
\frac{}{(chain_g f e) (flow_f v) \rightarrow (task_g (e v)) (flow_f v)} \\
\text{(R-FULFILFLOWVALUE)} \\
\frac{\neg isflow?(v)}{(task_f v) (flow_f) \rightarrow (flow_f v)} \\
\text{(R-CHAINVAL)} \\
\frac{\neg isflow?(v) \quad fresh g}{(task_f E[\mathbf{then*}(v, \lambda x.e)]) \rightarrow (flow_g) (task_g (\lambda x.e) v) (task_f E[g])} \\
\text{(R-FULFILFLOW)} \\
\frac{isflow?(g)}{(task_f g) \rightarrow (chain_f g \lambda x.x)} \\
\text{(R-CHAINFLOW)} \\
\frac{isflow?(h) \quad fresh g}{(task_f E[\mathbf{then*}(h, \lambda x.e)]) \rightarrow (flow_g) (chain_g h \lambda x.e) (task_f E[g])} \\
\text{(R-UNBOX)} \\
\frac{}{(task_f E[\mathbf{unbox} (\square v)]) \rightarrow (task_f E[v])}
\end{array}$$

■ **Figure 9** Run-time semantics.

as a small-step operational semantics with reduction-based, contextual rules for evaluation within tasks. Evaluation contexts  $E$  contain a hole  $\bullet$  that denotes the location of the next reduction step [40].

$$\begin{array}{l}
E ::= \bullet \mid E e \mid v E \mid \mathbf{return} E \mid \mathbf{get*} E \mid \mathbf{then*}(E, e) \mid \mathbf{then*}(v, E) \\
\quad \mid \square E \mid \mathbf{unbox} E \mid E [\tau]
\end{array}$$

The reduction rules (Figure 9) are the common  $\beta$ -reduction and type application from System F. The blocking operation  $\mathbf{get*} v$  performs a run-time check to test whether the value  $v$  is a data-flow future or simply a value lifted to one. If it is a data-flow future, the value is extracted (R-GETSTAR); in case of a value, it is left in place (R-GETVAL). Spawning a task creates a fresh data-flow future and task with a new task identifier, and the operation returns immediately the created future (R-AsyncStar). Returning from a task just throws away the execution context (R-RETURN), so that the task can fulfil its associated future in the next step. This next step depends on whether the value that fulfils the task is a future or a concrete value. If the task finishes with a data-flow future, the run-time chains the returned future to the identity function. This causes the value from the returned future to propagate to the current-in-call future (R-FULFILFLOW). If the return value of a task is not a data-flow future, then this simply fulfils the current-in-call future (R-FULFILFLOWVALUE). A chained configuration waits until the dependent data-flow future is fulfilled, then it executes the callback associated with it (R-CHAINRUNFLOW). Expression-level chaining on data-flow futures checks at run-time whether target of the chain operation on is a data-flow future or a lifted value. In the former case, it lifts the chaining from the expression to the configuration level, returning immediately a new data-flow future (R-CHAINFLOW). In the latter case, chaining creates a new task to apply the chained function (R-CHAINVAL). The reason for



spawning a new task is to preserve consistent behaviour across chaining on fulfilled and unfulfilled futures. If chaining on a fulfilled future executed immediately, and synchronously, we would increase the latency of the current task, or – if FlowFut is implemented in a language with mutable state – potentially introduce a race condition as it is unclear whether a chained lambda function executes directly (and synchronously) or not. This design saves a programmer from such potential hassles.

The unboxing operator unpacks the boxed value (R-UNBOX). It is important for encoding of control-flow futures in terms of data-flow futures, described in the upcoming section. Boxed values will be introduced in conjunction with the encoding.

**Extending FlowFut with Control-Flow Futures.** In this section we show how to extend the language with control-flow futures encoded in terms of data-flow futures. Operations on data-flow futures transparently traverse any number of (invisible-from-the-typing) nested data-flow futures until they reach a concrete value or a control-flow future. The inclusion of the boxed values allow us to straightforwardly encode **Fut**  $\tau$  thus: **Fut**  $\tau \hat{=} \square$  **Flow**  $\tau$ . Using this encoding, we extend FlowFut with equi-named operations on control-flow futures, dropping the  $\star$  for clarity. It is straightforward to encode each operation using its corresponding  $\star$ -version combined together with  $\square$  and **unbox**:

$$\mathbf{get} \ e \hat{=} \mathbf{get}\star(\mathbf{unbox} \ e) \quad \mathbf{then}(e, e') \hat{=} \square \mathbf{then}\star(\mathbf{unbox} \ e, e') \quad \mathbf{async} \ e \hat{=} \square \mathbf{async}\star \ e$$

A control-flow future is always a boxed value, where the value can be anything including another future (data-flow or control-flow), or a concrete value. To perform control-flow future operations, one always needs to unpack the box and use its equivalent data-flow future operator. When an operator returns a new control-flow future (chaining and spawning a task), the return value needs to be boxed again.

Similarly, we extend FlowFut with type rules for these operations. These are the same as their  $\star$ -versions except that they use control-flow future types. Chaining takes a control-flow future and a function acting as callback and returns immediately a new control-flow future (T-THEN). Spawning a task returns immediately a control-flow future (T-ASYNC). Blocking access on a control-flow future returns the value inside the future (T-GET).

$$\begin{array}{c} \text{(T-THEN)} \\ \frac{\Gamma \vdash_{\rho} e_1 : \mathbf{Fut} \ \tau' \quad \Gamma \vdash_{\rho} e_2 : \tau' \rightarrow \tau}{\Gamma \vdash_{\rho} \mathbf{then}(e_1, e_2) : \mathbf{Fut} \ \tau} \end{array} \quad \begin{array}{c} \text{(T-ASYNC)} \\ \frac{\Gamma \vdash_{\tau} e : \tau}{\Gamma \vdash_{\rho} \mathbf{async} \ e : \mathbf{Fut} \ \tau} \end{array} \quad \begin{array}{c} \text{(T-GET)} \\ \frac{\Gamma \vdash_{\rho} e : \mathbf{Fut} \ \tau}{\Gamma \vdash_{\rho} \mathbf{get} \ e : \tau} \end{array}$$

Because data-flow futures do not allow observing completion of individual stages of an operation returning a nested future, we design our system to always “forward-compress” the return value of a flow, meaning we treat **return** of data-flow futures implicitly as a **forward** from [13], which addresses the Future Proliferation Problem. This brings us to the final extension of FlowFut with support for **forward**. Forwarding a control-flow future is just unpacking it and returning it, whereas forwarding a data-flow future is equivalent to **return**:

$$\mathbf{forward} \ e \hat{=} \mathbf{return} \ (\mathbf{unbox} \ e) \quad \mathbf{forward}\star \ e \hat{=} \mathbf{return} \ e$$

And the type rules are straightforward: (Note that  $\tau'$  can be any well-formed type as the expression will not have a usual return type, but instead finish the enclosing task.)

$$\begin{array}{c} \text{(T-FORWARD)} \\ \frac{\Gamma \vdash \tau' \quad \Gamma \vdash_{\tau} e : \mathbf{Fut} \ \tau}{\Gamma \vdash_{\tau} \mathbf{forward} \ e : \tau'} \end{array} \quad \begin{array}{c} \text{(T-FORWARD-STAR)} \\ \frac{\Gamma \vdash \tau' \quad \Gamma \vdash_{\tau} e : \mathbf{Flow} \ \tau}{\Gamma \vdash_{\tau} \mathbf{forward}\star \ e : \tau'} \end{array}$$

► **Theorem** (Progress for FlowFut). *Given a global configuration  $config$ , if  $\Gamma \vdash config\ ok$ , then  $config$  is a terminal configuration or there exists a  $config'$  such that  $config \rightarrow config'$ .*

► **Theorem** (Preservation for FlowFut). *Given a global configuration  $config$ , if  $\Gamma \vdash config\ ok$  and  $config \rightarrow config'$ , then there exists a  $\Gamma'$  such that  $\Gamma' \supseteq \Gamma$  and  $\Gamma' \vdash config'\ ok$ .*

**Proof.** Both proofs are by induction on the derivation of the shape of  $config$ . ◀

We now move to FutFlow, which has control-flow futures as its primitive form of future. Section 5 revisits FlowFut and FutFlow, and discusses optimisation and implementation issues.

### 4.3 FutFlow: Primitive Control-Flow and Encoded Data-Flow Futures

A large number of programming languages implement control-flow explicit futures, natively (e.g., ABS [22], Encore [7], Joelle [10]) or through standard or third-party libraries (e.g., Java [32, 17], Akka [41]). In this section we explain, through the calculus FutFlow, how to extend such a semantic model to also encompass control-flow futures and delegation. Thus, in contrast to FlowFut, we now encode data-flow futures in terms of control-flow future types.

The calculus (omitting operations on data-flow futures) contains expressions and values:

$$\begin{aligned} e &::= v \mid ee \mid e[\tau] \mid \mathbf{return}\ e \mid \mathbf{async}\ e \mid \mathbf{get}\ e \mid \mathbf{then}(e, e) \mid \mathbf{forward}\ e \\ v &::= c \mid x \mid f \mid \lambda x.e \mid \lambda X.e \end{aligned}$$

The key difference to FlowFut is the inclusion of **forward** as a primitive.

**Static Semantics.** The type system has the following types: the common basic types ( $\mathcal{K}$ ), abstraction ( $\tau \rightarrow \tau$ ), type variables ( $X$ ), universal quantification ( $\forall X.\tau$ ), and control-flow future types (**Fut**  $\tau$ ).

$$\text{Types:} \quad \tau ::= \mathcal{K} \mid \tau \rightarrow \tau \mid X \mid \forall X.\tau \mid \mathbf{Fut}\ \tau$$

**Well-Typed Expressions.** The type formation rules are given in Figure 10 and expression typing is shown in Figure 11 – similar to Section 4.2. Most rules should be straightforward and have appeared before in similar form. Note lack of  $\star$  on operators to highlight the control-flow nature.

$$\begin{array}{c} \frac{}{\vdash \epsilon} \text{(TF-ENV)} \qquad \frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau} \text{(TF-ENVEXPR)} \qquad \frac{X \notin \text{dom}(\Gamma) \quad \vdash \Gamma}{\vdash \Gamma, X} \text{(TF-ENVVAR)} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{K}} \text{(TF-K)} \\ \\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathbf{Fut}\ \tau} \text{(TF-FUT)} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash \tau \rightarrow \tau'} \text{(TF-ARROW)} \qquad \frac{X \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash X} \text{(TF-X)} \qquad \frac{\Gamma, X \vdash \tau}{\Gamma \vdash \forall X.\tau} \text{(TF-FORALL)} \end{array}$$

■ **Figure 10** Type formation rules where  $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, X$ .

$$\begin{array}{c}
\text{(T-CONSTANT)} \\
\frac{c \text{ has type } \mathcal{K} \quad \Gamma \vdash \mathcal{K}}{\Gamma \vdash_{\rho} c : \mathcal{K}} \\
\\
\text{(T-VARIABLE)} \\
\frac{x : \tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_{\rho} x : \tau} \\
\\
\text{(T-FUT)} \\
\frac{f : \mathbf{Fut} \tau \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash_{\rho} f : \mathbf{Fut} \tau} \\
\\
\text{(T-RETURN)} \\
\frac{\Gamma \vdash_{\tau} e : \tau \quad \tau \neq \bullet \quad \Gamma \vdash \tau'}{\Gamma \vdash_{\tau} \mathbf{return} e : \tau'} \\
\\
\text{(T-ABSTRACTION)} \\
\frac{\Gamma, x : \tau \vdash_{\bullet} e : \tau'}{\Gamma \vdash_{\rho} \lambda x. e : \tau \rightarrow \tau'} \\
\\
\text{(T-APPLICATION)} \\
\frac{\Gamma \vdash_{\rho} e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash_{\rho} e_2 : \tau}{\Gamma \vdash_{\rho} e_1 e_2 : \tau'} \\
\\
\text{(T-FORWARD)} \\
\frac{\Gamma \vdash \tau' \quad \tau \neq \bullet \quad \Gamma \vdash_{\tau} e_1 : \mathbf{Fut} \tau}{\Gamma \vdash_{\tau} \mathbf{forward} e_1 : \tau'} \\
\\
\text{(T-TYPEABSTRACTION)} \\
\frac{\Gamma, X \vdash_{\bullet} e : \tau}{\Gamma \vdash_{\rho} \lambda X. e : \forall X. \tau} \\
\\
\text{(T-TYPEAPPLICATION)} \\
\frac{\Gamma, X \vdash_{\rho} e : \forall X. \tau'}{\Gamma \vdash_{\rho} e [\tau] : \tau' [\tau/X]} \\
\\
\text{(T-THEN)} \\
\frac{\Gamma \vdash_{\rho} e_1 : \mathbf{Fut} \tau' \quad \Gamma \vdash_{\tau} e_2 : \tau' \rightarrow \tau}{\Gamma \vdash_{\rho} \mathbf{then}(e_1, e_2) : \mathbf{Fut} \tau} \\
\\
\text{(T-ASYNC)} \\
\frac{\Gamma \vdash_{\tau} e : \tau}{\Gamma \vdash_{\rho} \mathbf{async} e : \mathbf{Fut} \tau} \\
\\
\text{(T-GET)} \\
\frac{\Gamma \vdash_{\rho} e : \mathbf{Fut} \tau}{\Gamma \vdash_{\rho} \mathbf{get} e : \tau}
\end{array}$$

■ **Figure 11** Typing of expressions.

$$\begin{array}{c}
\text{(R-}\beta\text{)} \\
(task_f E[\lambda x. e v]) \rightarrow (task_f E[e[v/x]]) \\
\\
\text{(R-ASYNC)} \\
\frac{fresh f}{(task_g E[\mathbf{async} e]) \rightarrow (fut_f) (task_f e) (task_g E[f])} \\
\\
\text{(R-CHAINRUNFUT)} \\
(chain_g f e) (fut_f v) \rightarrow (task_g (e v)) (fut_f v) \\
\\
\text{(R-GET)} \\
(task_f E[\mathbf{get} h]) (fut_h v) \rightarrow (task_f E[v]) (fut_h v) \\
\\
\text{(R-FUTFULFILVALUE)} \\
(task_f v) (fut_f) \rightarrow (fut_f v) \\
\\
\text{(R-TYPEAPPLICATION)} \\
(task_f E[(\lambda X. e) [\tau]]) \rightarrow (task_f E[e[\tau/X]]) \\
\\
\text{(R-FORWARD)} \\
(task_f E[\mathbf{forward} h]) \rightarrow (chain_f h \lambda x. x) \\
\\
\text{(R-RETURN)} \\
(task_f E[\mathbf{return} v]) (fut_f) \rightarrow (fut_f v) \\
\\
\text{(R-THEN)} \\
\frac{fresh g}{(task_f E[\mathbf{then}(h, \lambda x. e)]) \rightarrow (fut_g) (chain_g h \lambda x. e) (task_f E[g])}
\end{array}$$

■ **Figure 12** Run-time semantics.

**Operational semantics.** The operational semantics are similar to Section 4.2. Evaluation contexts  $E$  contain a hole  $\bullet$  that denotes where the next reduction step happens [40]:

$$E ::= \bullet \mid E e \mid v E \mid \mathbf{then}(E, e) \mid \mathbf{forward} E \mid E [\tau] \mid \mathbf{get} E \mid \mathbf{return} E$$

The reduction rules are similar to the FlowFut calculus, but work on control-flow futures (Figure 12). Beta reduction works in the traditional fashion. The **async** construct spawns a new task to execute the given expression, and creates a new control-flow future to store its result (R-ASYNC). A chained configuration runs as soon as the dependent future is fulfilled and passes the content of the fulfilled future to the callback expression, running the pending computation on demand (R-CHAINRUNFUT). Getting a value out of a future blocks the execution until the future is fulfilled (R-GET). Tasks fulfil their implicit future implicitly, when there are no more pending expressions to run, or explicitly via the return expression

(R-FUTUREFULFILLVALUE and R-RETURN). So far, most **forward** examples have avoided future nesting by reusing the current-in-call future with a following asynchronous operation. This avoids creation of an additional future, meaning nesting is not possible. It is also possible to use **forward** to fulfil one existing future with the result of another without nesting or blocking the current computation: **forward**  $h$  fulfils the current-in-call future with the value in  $h$  by throwing away the remainder of the body of the current task and chaining the identity function on  $h$ . This has the effect of copying the eventual result stored in  $h$  into the current future (R-FORWARD). 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 (R-THEN).

**Extending FutFlow with Data-Flow Futures.** In this section we show, first, the language extensions necessary for encoding data-flow futures in terms of control-flow futures and, second, the encodings.

We extend the calculus with the following expressions and values:

$$e ::= \dots \mid \mathbf{match}(x : e, x : e, e) \mid \blacklozenge e \qquad v ::= \dots \mid \blacklozenge v$$

Expressions can now use a pattern matching operation, which is a common programming construct [31, 24]. To encode data-flow futures, we define a boxing operation ( $\blacklozenge e$ ) which uses pattern matching for unboxing. To keep constructs simple, **match** and  $\blacklozenge$  are not intended as source-level constructs, so  $\blacklozenge$  only works on data-flow futures in the formalism.

The type system has the previous common types with the addition of the  $\blacklozenge$  type – used later for encoding data-flow futures in terms of control-flow futures. As in FlowFut, we show explicit flattening rules for the encodings of data-flow futures:

$$\begin{array}{l} \text{Types:} \\ \text{Previous flattening rules and:} \end{array} \qquad \begin{array}{l} \tau ::= \dots \mid \blacklozenge \mathbf{Fut} \tau \\ \downarrow \blacklozenge \mathbf{Fut} (\blacklozenge \mathbf{Fut} \tau) = \downarrow \blacklozenge (\mathbf{Fut} \tau) \\ \downarrow \blacklozenge \mathbf{Fut} \tau = \blacklozenge (\downarrow \mathbf{Fut} \tau) \quad \tau \neq \blacklozenge \mathbf{Fut} \tau' \end{array}$$

Additional type rules for these constructs are found below. Notice how the introduction of the explicit flattening rules require the update of two typing rules (T-TYPEABSTRACTION and T-TYPEAPPLICATION). This is necessary to flatten  $\blacklozenge \mathbf{Fut} \tau$  types.

$$\begin{array}{c} \text{(DIA)} \qquad \text{(T-TYPEABSTRACTION)} \qquad \text{(T-TYPEAPPLICATION)} \\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \blacklozenge \tau} \qquad \frac{\Gamma, X \vdash \bullet e : \tau}{\Gamma \vdash_{\rho} \lambda X. e : \downarrow \forall X. \tau} \qquad \frac{\Gamma, X \vdash_{\rho} e : \downarrow \forall X. \tau'}{\Gamma \vdash_{\rho} e [\tau] : \downarrow \tau' [\tau/X]} \\ \\ \text{(T-MATCH)} \qquad \text{(T-VALFLOW)} \qquad \text{(T-FLOWFUT)} \\ \frac{\Gamma, x : \tau \vdash_{\rho} e_1 : \tau' \qquad \Gamma \vdash_{\rho} e_3 : \downarrow \blacklozenge \mathbf{Fut} \tau}{\Gamma \vdash_{\rho} \mathbf{match}(x : e_1, x : e_2, e_3) : \tau'} \qquad \frac{\Gamma \vdash_{\rho} e : \tau}{\Gamma \vdash_{\rho} e : \downarrow \blacklozenge \mathbf{Fut} \tau} \qquad \frac{\Gamma \vdash_{\rho} e : \mathbf{Fut} \tau}{\Gamma \vdash_{\rho} \blacklozenge e : \downarrow \blacklozenge \mathbf{Fut} \tau} \end{array}$$

The **match** construct has two open terms as first and second arguments, the free variables are captured at the declaration site; the third argument is a data-flow future type argument. The first argument is applied if the data-flow future type is actually a value and the second argument is applied to the value of the data-flow future type if the type was lifted from a control-flow future. Essentially, **match** pattern matches on the form of the data-flow future type. An expression of type  $\tau$  can be lifted to  $\blacklozenge \mathbf{Fut} \tau$  (T-VALFLOW and T-FLOWFUT).

The dynamic semantics include now the pattern matching operation, which performs beta reduction based on the form of the value  $v$  (R-MATCH-VAL and R-MATCH-FUT).

The introduction of the boxing value – used to encode data-flow futures – requires special care when fulfilling of a task. This is reflected in the updated rule R-FUTFULFILVALUE and on R-FLOWCOMPRESSION. If the value is not a data-flow future, i.e.,  $v \neq \blacklozenge v'$ , then the value fulfils the task's future; if the value is a data-flow future, then it builds a chained configuration to ultimately pull the value out, running the identity function.

$$\frac{\text{(R-FUTFULFILVALUE)} \quad v \neq \blacklozenge v'}{(task_f v) (fut_f) \rightarrow (fut_f v)} \quad \text{(R-MATCHVAL)} \quad (task_f E[\mathbf{match}(x : e_1, x : e_2, v)]) \rightarrow (task_f E[e_1[v/x]])$$

$$\text{(R-FLOWCOMPRESSION)} \quad (task_f \blacklozenge g) \rightarrow (chain_f g \lambda x.x) \quad \text{(R-MATCHFUT)} \quad (task_f E[\mathbf{match}(x : e_1, x : e_2, \blacklozenge g)]) \rightarrow (task_f E[e_2[g/x]])$$

With these new constructs, we can encode data-flow futures in terms of control-flow futures:  $\mathbf{Flow} \tau \hat{=} \blacklozenge \mathbf{Fut} \tau$ . The term  $\blacklozenge e$  captures the lifting of a control-flow future value to  $\mathbf{Flow} \tau$  (T-FLOWFUT). All operators on data-flow futures are encoded in terms of primitive operators:

$$\begin{aligned} \mathbf{async}^* e &\hat{=} \blacklozenge \mathbf{async} e & \mathbf{then}^*(e, fn) &\hat{=} \mathbf{match}(x : fn x, f : \blacklozenge \mathbf{then}(f, fn), e) \\ \mathbf{get}^* e &\hat{=} \mathbf{match}(x : x, x : \mathbf{get} x, e) & \mathbf{forward}^* e &\hat{=} \mathbf{match}(x : \mathbf{return} x, f : \mathbf{forward} f, e) \\ & & \mathbf{undiamond} e &\hat{=} \mathbf{get}^* e \end{aligned}$$

The typing rules for operations on data-flow futures are expressed as an extension to the typing rules of Figure 11. A data-flow future type can be “unlifted” so that we extract its internal value (T-UNDIAMOND). Any expression can be lifted from some  $\tau$  or from a control-flow future type to a data-flow future (rules T-FLOW and T-FLOWFUT).

$$\begin{array}{c} \text{(T-ASYNC-STAR)} \\ \frac{\Gamma \vdash_\rho e : \tau}{\Gamma \vdash_\rho \mathbf{async}^* e : \downarrow \mathbf{Flow} \tau} \end{array} \quad \begin{array}{c} \text{(T-GET-STAR)} \\ \frac{\Gamma \vdash_\rho e : \downarrow \mathbf{Flow} \tau}{\Gamma \vdash_\rho \mathbf{get}^* e : \tau} \end{array}$$

$$\begin{array}{c} \text{(T-THEN-STAR)} \\ \frac{\Gamma \vdash_\rho e_1 : \downarrow \mathbf{Flow} \tau' \quad \Gamma \vdash_\rho e_2 : \tau' \rightarrow \tau}{\Gamma \vdash_\rho \mathbf{then}^*(e_1, e_2) : \downarrow \mathbf{Flow} \tau} \end{array} \quad \begin{array}{c} \text{(T-FORWARD-STAR)} \\ \frac{\Gamma \vdash_\tau e : \downarrow \mathbf{Flow} \tau}{\Gamma \vdash_\tau \mathbf{forward}^* e : \tau'} \end{array} \quad \begin{array}{c} \text{(T-UNDIAMOND)} \\ \frac{\Gamma \vdash_\rho e : \downarrow \blacklozenge \mathbf{Fut} \tau}{\Gamma \vdash_\rho \mathbf{undiamond} e : \tau} \end{array}$$

This concludes the presentation of FutFlow. In the next section, we discuss optimisations in FlowFut and FutFlow, and implementation issues.

► **Theorem** (Progress for FutFlow). *Given a global configuration  $config$ , if  $\Gamma \vdash config$  ok, then  $config$  is a terminal configuration or there exists a  $config'$  such that  $config \rightarrow config'$ .*

► **Theorem** (Preservation for FutFlow). *Given a global configuration  $config$ , if  $\Gamma \vdash config$  ok and  $config \rightarrow config'$ , then there exists a  $\Gamma'$  such that  $\Gamma' \supseteq \Gamma$  and  $\Gamma' \vdash config'$  ok.*

**Proof.** Both proofs are by induction on the derivation of the shape of  $config$ . ◀

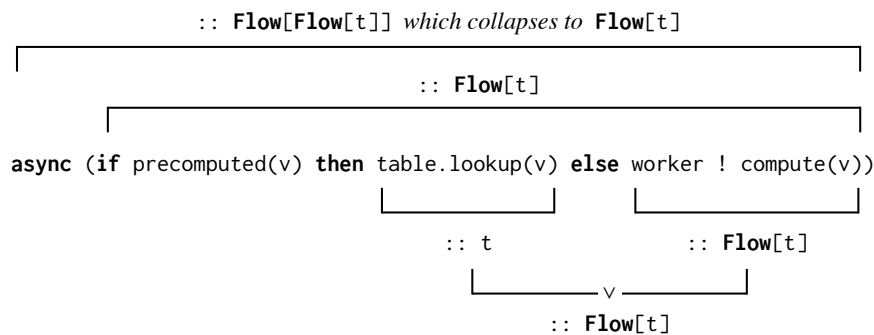
## 4.4 Godot's Solutions to Future Problems

We now revisit the problems of Section 2 and show how Godot addresses them.

**The Type Proliferation Problem.** Because of the data-flow future component, the Type Proliferation Problem is avoided. Like with DeF, the following statement is typeable (as is tail-recursive functions) and returns a `Flow[t]` (the equivalent of `Flow τ` in code examples):

```
async (if precomputed(v) then table.lookup(v) else worker ! compute(v))
```

Because data-flow futures allow implicitly lifting a concrete value of type `t` to `Flow[t]`, code using data-flow futures can be trivially reused with concrete values. This addresses the Type Proliferation Problem, allowing one data-flow future type to represent zero or many run-time futures. (See Figure 13 for additional details.)



■ **Figure 13** Overcoming the Type Proliferation Problem. Compare with Figure 2.

**The Future Proliferation Problem.** Because of support for delegation, the Future Proliferation Problem is avoided – but in a way that improves on `forward`. Since data-flow futures abstract nesting, we can implicitly turn a `return` into a `forward` based on the return type, and not require the programmer to explicitly choose a forwarding solution. Thus, we can avoid the quirky looking `return` in one branch and `forward` in another, and simply write:

```
if precomputed(v) then return table.lookup(v) else return worker ! compute(v)
```

This allows us to hoist the `return` to write the original statement that would not type with explicit control-flow futures, while still avoiding creation of unnecessary futures:

```
return if precomputed(v) then table.lookup(v) else worker ! compute(v)
```

**The Fulfilment Observation Problem.** The integration of both kinds of futures in a single system avoids the Fulfilment Observation Problem by allowing a programmer to opt-in on control-flow futures where desirable, without imposing a one-size-fits-all solution. The following function definition uses explicit control-flow futures to allow the observation of both stages – finding an idle worker and dispatching work to it and completing the job:

```
def perform(job : Job[t]) : Fut[t] { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job) --f is a control-flow future
get(f) --block until do_work() has been called
```

In contrast, this function definition uses data-flow futures and therefore will *not* allow the distinction between the two stages, and its **return** will be treated as a **forward**:

```
def perform(job : Job[t]) : Flow[t] { return idle_worker() ! do_work(job) }
var f = load_balancer ! perform(my_job) --f is a data-flow future
get(f) --block until do_work() has finished
```

Notably, the integrated system also supports the nesting of *different kinds* of futures. For example,  $\mathbf{Flow}[\mathbf{Fut}[\mathbf{Flow}[t]]]$  denotes a value computed by a pipeline of zero or more asynchronous operations whose individual completedness cannot be distinguished, followed by a control-flow future corresponding to a single operation *whose completedness can be observed*, followed by another pipeline of zero or more asynchronous operations.

**Concluding Remarks.** In addition to addressing all three problems of Section 2, Godot overcomes a limitation in the initial DeF proposal for data-flow explicit futures in [20] by adding support for *parametric polymorphism*. In fact, DeF did not study parametric polymorphism and it is not trivial to add, as standard techniques [33] prevent the collapsing of nested future types. For example, in DeF the following function  $\mathbf{problematic} = (\lambda X.\lambda y : X.\mathbf{async}^* y)$  has type  $\forall X.X \rightarrow \mathbf{Flow} X$  and, after type application  $\mathbf{problematic} [\mathbf{Flow} \mathcal{K}] :: \mathbf{Flow}(\mathbf{Flow} \mathcal{K})$ , which forces a programmer to insert multiple **get** operations to obtain a concrete value from a data-flow future, which breaks the DeF invariant that a single **get** is always enough to access a concrete value. In Godot, the **problematic** function after type application has type  $\mathbf{Flow} \mathcal{K}$ , because typing rules normalise flow types and **get\*** guarantees access to a concrete value.

Using Godot, a programmer can decide to abstract or expose details about how values are produced through asynchronous operations, by freely choosing between control-flow futures and data-flow futures or any combination thereof. And in the case of data-flow futures, profit from how Godot automatically avoids creating unnecessary (unobservable) futures. As the integration of control-flow futures, delegation, and data-flow futures improves the individual components (e.g., the support for parametric polymorphism with data-flow futures and type-driven automatic insertion of **forward**), *Godot is greater than the sum of its parts*. Moreover, as the next sections will show, it is possible to encode either kind of future in the other, which facilitates their implementation in a programming language. This realisation is an important aspect of our contributions, which extends beyond “taking the union.”

This section has put in perspective Godot as solution to the Type Proliferation Problem, Future Proliferation Problem, and Fulfilment Observation Problem through the integration of control-flow futures and data-flow futures in an explicit system with support for implicit delegation. The previous sections 4.1–4.3 explain Godot in detail.

## 5 Discussion

The preceding two sections showed how to encode data-flow futures in a language that only provides control-flow futures, and the opposite; both approaches rely on small extensions of the type system and encodings of operations for one type of futures into the other. We review below the preceding results from an implementation and optimisation point of view.

### 5.1 Avoiding Future Nesting through Implicit Delegation

We revisit the example from the introduction to the Fulfilment Observation Problem (Section 2). We imagine that this method runs in the context of an actor that “load-balances” by

farming out jobs to the worker returned from the `idle_worker()`. As discussed previously, this is a case where control-flow explicit futures insert an additional, possibly unwanted, future indirection due to the additional asynchronous call handing the work off to some worker.

```
def perform(job : Job[t]) : Flow[t] { return idle_worker() ! do_work(job) }
```

as the programmer declared the return type of `perform()` as `Flow[t]`, the implementation is less restricted (e.g., we can either delegate to a worker or return a cached result without the typing problems of Figure 2). The programmer is not interested in any intermediate stages of the computation, and we can compile the method body replacing `return` with `forward`. *This optimisation is crucial for making asynchronous tail-recursive calls run in constant space.*

We model this example and optimisation in `FlowFut` (or `FutFlow`) as if the body of `perform()` executes inside a task, whose final expression is a `return async*` with the body of `do_work()` inside it. We express this optimisation in `FlowFut` as follows (rule `RETURNASYNC`):

$$\frac{\text{(RETURNASYNC)} \quad \text{fresh } j}{(task_f^i E[\mathbf{return\ async*} e]) \rightarrow (task_f^j e)} \quad \frac{\text{(RETURNTHEN)} \quad isflow?(g)}{(task_f^i E[\mathbf{return\ then*}(g, e)]) \rightarrow (chain_f g e)}$$

For clarity, we add identifiers  $i$  and  $j$  to the tasks to highlight that there is *no reuse of a task* (which models delegating work to another concurrent actor), but a *reuse of a future* (the semantics of `forward`). We apply a similar optimisation (rule `RETURNTHEN`) when returning the result of future chaining: task  $i$  delegates the fulfilment of  $f$  to the chain task, and the delegating task finishes (and is removed in the calculus).

**How Nesting Causes False Fulfilment.** As exemplified in the previous section, implicit delegation avoids the creation of nested futures. We now illustrate why false fulfilment can happen if we do not avoid future nesting. Consider the implementation of `get*` and suppose we had a non-optimised version of `FlowFut` that has nesting of futures. Formally, we would have a reduction rule –  $(flow_f) (task_f g) \rightarrow (flow_f g)$  – that fulfils a data-flow future with another data-flow future. As a consequence, `get*` must perform a run-time test, and branch on whether a future is fulfilled by a concrete value or another future:

$$\frac{\text{(R-GETSTARFLOW-UNOPT)} \quad isflow?(g)}{(task_f E[\mathbf{get*} g]) (flow_g v) \rightarrow (task_f E[\mathbf{get*} v]) (flow_g v)} \quad \frac{\text{(R-GETSTARVAL-UNOPT)} \quad \neg isflow?(g)}{(task_f E[\mathbf{get*} v']) \rightarrow (task_f E[v'])}$$

The key rule above is `R-GETSTARFLOW-UNOPT` which shows that a `get*` yielding a future reduces to another `get*`, meaning we move to another possibly blocked state. This does not happen in `FlowFut`. Indeed, if a task returns a data-flow future  $g$  in a way that delegation could not elide, by `R-FULFILFLOWVALUE` we use future chaining on  $g$  and tell  $g$  to fulfil  $f$  on its fulfilment instead having  $f$  effectively polling  $g$  through the implementation of `get*`.

## 5.2 Notes on Implementing Godot

Let us first consider the encodings. Implementing the encoding rules, either as a compilation phase or even as a library is pretty straightforward except from the following points.

1. Both encodings rely on the existence of a boxing construct. Such a construct can be easily encoded with a datatype or an object type. However due to the simplicity of the operations on boxes a native implementation could be more efficient.



2. The encoding of data-flow futures from control-flow ones requires a pattern matching operator (or equivalent) that can distinguish data-flow futures from other values, unless lifting actually creates a fulfilled data-flow future. Standard compiler optimisations are applicable here, such as synthesising different methods from a single specification, e.g., one applies only to values where future types are removed, one applies only to statically verified actual data-flow futures, and one for all other cases, or combinations.

Second, consider the type system extensions. In both cases, the type system of the language can be extended with the existing rules without major difficulty. Additionally, without modifying the type system if the data-flow future language has parametric types it seems possible to encode control-flow future typing rules: all typing rules necessary to extend the data-flow future language can be expressed as typing of parametric types and functions. In the other direction, it is slightly more involved as the type system of the control-flow future language must implement a form of type collapse rule.

Overall, extending a language with data-flow (resp. control-flow) futures to also support control-flow (resp. data-flow) futures raises no particular difficulty, and the encodings avoid adding “native support” for both futures. The compiler and the type checker need a small number of new, simple constructs. A data-flow future language may have to add chaining on data-flow futures, or the control-flow future language should add pattern matching that distinguishes data-flow futures. While all these extensions are minor, they will require some modicum of language modifications. Consequently, the implementation of Godot as an external library of a mainstream language is not straightforward: standard type systems do not perform the implicit lifting required for data-flow futures, and the future chaining required for data-flow futures rarely exist in control-flow futures. However, if a language with data-flow futures already supports chaining, which is a common operation, implementing control-flow futures in a non-intrusive manner – as an external library – seems to raise no difficulty.

## 6 Related Work

Section 3 discusses the most closely related work on data-flow explicit futures (DeF) [20] and future delegation [13] in detail. Earlier work on adapting a static analysis [16] from explicit to implicit futures revealed the difference between the future accesses, and a translation of control-flow synchronisation in ABS [22] to data-flow synchronisation in ProActive [21] showed that control-flow synchronisation could not be simulated purely by implicit futures.

Futures are means for expressing concurrency while enabling synchronisation at the latest possible time. They were first introduced by Baker and Hewitt in the 70’s [4], and later rediscovered by Liskov and Shriram as Promises [27] and by Halstead in the context of MultiLisp [25]. Flanagan and Felleisen did an early formalisation of futures [15] based on MultiLisp’s futures with focus on the difference between explicit and implicit future access. In a similar vein,  $\lambda(fut)$  [29] is a concurrent lambda calculus with futures with cells and handles. Futures in  $\lambda(fut)$  are explicitly created, similarly to MultiLisp. We now consider futures with respect to the dichotomy between implicit and explicit futures.

**Implicit Futures.** Implicit futures are indistinguishable from concrete values in source code. Typically, data-flow synchronisation is based on implicit futures. In MultiLisp [25], the future construct creates a thread and returns a future that can be manipulated by operations like assignment, that do not need a real value, but the program would *automatically block* when performing an operation requiring the value, i.e., futures are implicitly accessed but explicitly

created. Similar constructs can be found in Alice [35], Oz-Mozart [38], and ProActive [3]. The latter is a Java library for active objects in which futures are implicit like in MultiLisp; futures are implemented with proxies that hide the future and provide a normal object interface, but accessing a proxy leads to a synchronisation with the availability of the object, i.e., the fulfilment of the future.

**Explicit Futures.** Explicit, typed futures appeared with ABCL/f [37] to represent the result of an asynchronous method invocation, paving the way for active object languages [12]. Explicit futures typically have a parametric future type, and exist in, e.g., Concurrent ML [34], C++ [26] and Java [39], often as libraries. Explicit futures open for different ways of synchronising. Hybrid [30] is an early language with *forwarding*; in this paper, Nierstrasz formulates a version of the *Future Proliferation Problem* of Section 2. Creol [23] features non-blocking polling on futures that enables *cooperative multi-threading* based on future availability. De Boer et al. [11] were probably among the first to offer a rich set of future manipulation primitives for control flow, together with a compositional proof theory. JCoBox [36], ABS [22], and Encore have mostly reused these primitives [36, 22, 7]. Encore additionally supports creation and manipulation of parallel tasks as sets of futures [14].

Akka [19, 41] is a scalable actor library implemented on top of Java and Scala, in which futures are used either to allow actor messages to return a value or more automatically in the messages of *typed actors* (akin to active objects). The Akka programmer is advised to use asynchronous reaction on futures, i.e., register code to be executed when the future is fulfilled. Akka's `map` construct is similar to our `then` chaining construct. JavaScript promises are data-flow synchronised futures with explicit asynchronous access and no typing. The data-flow nature of the synchronisation distinguishes JavaScript from the other languages with explicit futures and is probably related to the absence of future type. Because it is untyped and promises are explicitly accessed and fulfilled, errors are frequently made when manipulating these promises; Madsen et al. [28] provide a powerful tool to study these errors.

**Futures in the Mainstream.** Many modern, statically typed programming languages provide control-flow futures through libraries to facilitate the creation and control of asynchronous, concurrent computations. We highlight Completable Futures [32], Listenable Futures [17], Scala Futures [18] and Akka Futures [1], and the **Observable** abstraction from the ReactiveX library [2], where asynchronous computations may return (*emit*) more than one value.

Future libraries of mainstream languages have considerably richer interfaces than the future abstractions in our core calculus, but, as far as we can tell, we provide all the necessary operations to construct most, if not all, of these library interfaces.

Extending existing libraries with support for data-flow futures is an interesting direction of future work. We take some preliminary steps in this direction in the companion artefact of this paper which shows how to integrate data-flow futures with Scala futures.

There is an analogy between futures and the observable abstraction from the ReactiveX library: both are control-flow constructs. Investigating whether the benefits of data-flow futures can be carried over to observables is an interesting future direction of this work.

Finally, most future libraries establish futures as monads, such as Akka Futures or the ReactiveX library. The control-flow futures from this paper are monadic, with `async` as its unit and `get` as its `join`. Data-flow futures are monadic as well, although they work on a smaller set of types, due to their implicit nature, i.e., they collapse **Flow** types.

## 7 Conclusion

The distinction between implicit and explicit futures is well-known, but recent work highlights that the relation between the typing and synchronisation discipline plays a more crucial aspect.

Following this observation, we identified three problems with existing future implementations: the Type Proliferation Problem restricts the expressiveness of control-flow futures; the Fulfilment Observation Problem limits the synchronisation capacities of data-flow futures; the Future Proliferation Problem makes both data-flow and control-flow futures inefficient. This paper defines Godot, a system supporting *both* data-flow and control-flow futures simultaneously, and in combination; our system is the first to do so, and also to solve the three problems above coherently in a single programming model. Godot shows how to add parametric polymorphism and automatic delegation for data-flow explicit futures, and demonstrates how to encode each type of future in terms of the other. This facilitates implementation of the full Godot system, or subsets, in existing programming languages, with or without support for futures. We believe that our formalisms communicate the core ideas, while not tying ourselves too closely to one particular kind of language or unit of concurrency.

While we developed two possible encodings, starting from a data-flow language seems a bit more promising. Indeed, if data-flow futures are the default, the non-expert programmer is only exposed to futures that do not suffer from Type Proliferation and where Future Proliferation can be avoided automatically. Programs that need control on future synchronisation, e.g., to implement load balancing or scheduling features, can use the encoding of control-flow futures and avoid the Fulfilment Observation Problem.

---

## References

- 1 Akka Futures. <https://doc.akka.io/docs/akka/current/futures.html>, 2019.
- 2 Rx Extensions. <http://reactivex.io/>, 2019.
- 3 Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Programming, Composing, Deploying for the Grid*, pages 205–229. Springer London, London, 2006.
- 4 Henry G. Baker and Carl E. Hewitt. The Incremental Garbage Collection of Processes. In *Proc. Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.
- 5 Samuel Beckett. Waiting for Godot. *Samuel Beckett: The Complete Dramatic Works*, pages 7–89, 1954.
- 6 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996. doi:10.1006/jpdc.1996.0107.
- 7 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 *Advanced Lectures on Formal Methods for Multicore Programming - 15th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2015)*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.
- 8 Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.
- 9 Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Tobias Wrigstad, and Albert Mingkun Yang. Attached and Detached Closures in Actors. In *Proc. 8th ACM SIGPLAN Intl. Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2018, pages 54–61. ACM, 2018. doi:10.1145/3281366.3281371.

- 10 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal Ownership for Active Objects. In G. Ramalingam, editor, *Proc. 6th Asian Symposium on Programming Languages and Systems (APLAS 2008)*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2008.
- 11 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.
- 12 Frank S. 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, 2017. doi:10.1145/3122848.
- 13 Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. Forward to a Promising Future. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Proc. 20th IFIP WG 6.1 Intl. Conf. on Coordination Models and Languages (COORDINATION 2018)*, volume 10852 of *Lecture Notes in Computer Science*, pages 162–180. Springer, 2018.
- 14 Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. ParT: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch-Lafuente and José Proenca, editors, *Proc. 18th IFIP WG 6.1 Intl. Conf. on Coordination Models and Languages (COORDINATION 2016)*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016. doi:10.1007/978-3-319-39519-7\_7.
- 15 Cormac Flanagan and Matthias Felleisen. The Semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.
- 16 Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *PPDP 2016 18th International Symposium on Principles and Practice of Declarative Programming*, Edinburgh, United Kingdom, September 2016. URL: <https://hal.inria.fr/hal-01345315>.
- 17 Google. Listenable Future Explained. <https://github.com/google/guava/wiki/ListenableFutureExplained>, January 2018.
- 18 Philipp Haller, Heather Miller, Aleksandar Prokopec, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises. <http://docs.scala-lang.org/overviews/core/futures.html>, 2016.
- 19 Phillip Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
- 20 Ludovic Henrio. Data-flow Explicit Futures. Research report, I3S, Université Côte d’Azur, April 2018. URL: <https://hal.archives-ouvertes.fr/hal-01758734>.
- 21 Ludovic Henrio and Justine Rochas. Multiactive objects and their applications. *Logical Methods in Computer Science*, Volume 13, Issue 4, November 2017. doi:10.23638/LMCS-13(4:12)2017.
- 22 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer Verlag, 2011.
- 23 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. doi:10.1016/j.tcs.2006.07.031.
- 24 Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- 25 Robert H. Halstead Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. doi:10.1145/4472.4478.
- 26 R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proc. Pattern Languages of Programs*, 1995.

- 27 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. doi:10.1145/53990.54016.
- 28 Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017. doi:10.1145/3133910.
- 29 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.
- 30 Oscar Nierstrasz. Active Objects in Hybrid. In Norman K. Meyrowitz, editor, *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 243–253. ACM, 1987.
- 31 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.
- 32 Oracle. JDK 10 for `java.util.concurrent.Future`. <https://docs.oracle.com/javase/10/docs/api/index.html?java/util/concurrent/Future.html>, 2018.
- 33 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- 34 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
- 35 Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklau, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, February 2006.
- 36 Jan Schafer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.
- 37 Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.
- 38 Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.
- 39 Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.
- 40 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.
- 41 Derek Wyatt. *Akka Concurrency*. Artima, 2013.