Programming Safe Agents in Blueprint

Alex Muscar
University of Craiova

WIMS'14

Programmers are craftsmen, and, as such, they are only as productive as theirs tools allow them to be

Introduction

Agent Oriented Programming

- has been around for 20+ years
- was intended as a higher level alternative to OOP
- many regarded it as “a revolution in software”

Agent Oriented Programming

- has failed to gain wide traction
- is regarded as an experimentation tool for AI
- the community lacks focus

Blueprint

Premise: There is still place for a solution that is both high-level, yet practical

Design goals: an agent oriented programming language, focusing on concurrency, static safety, ease of use and extensibility

Defining terms

agents: computational entities that (i) have their own thread of control and can decide autonomously if and when to perform a given action; and (ii) communicate with other agents by asynchronous message passing

Defining terms

concurrency: the composition of independently executing entities

Defining terms

consistency: data consistency*, rather than logical consistency

*The ‘A’ in A.C.I.D.

Defining terms

scalability: (i) the ability of the runtime to gracefully handle a growing number of agents executing concurrently; and (ii) the ability of the language to gracefully handle growing code bases

Background and Motivation

Why another language?

Blueprint's development was motivated by my experience:

- developing the prototype of a dynamic negotiation mechanism in Jason
- teaching agent technologies to undergraduate students using JADE

Jason’s advantages

- high level
- domain oriented
- most popular AOPL

Jason’s advantages

- active community
- regularly updated
- good documentation

Jason’s disadvantages

- limited in scope
- latently typed
- exotic syntax

Jason’s disadvantages

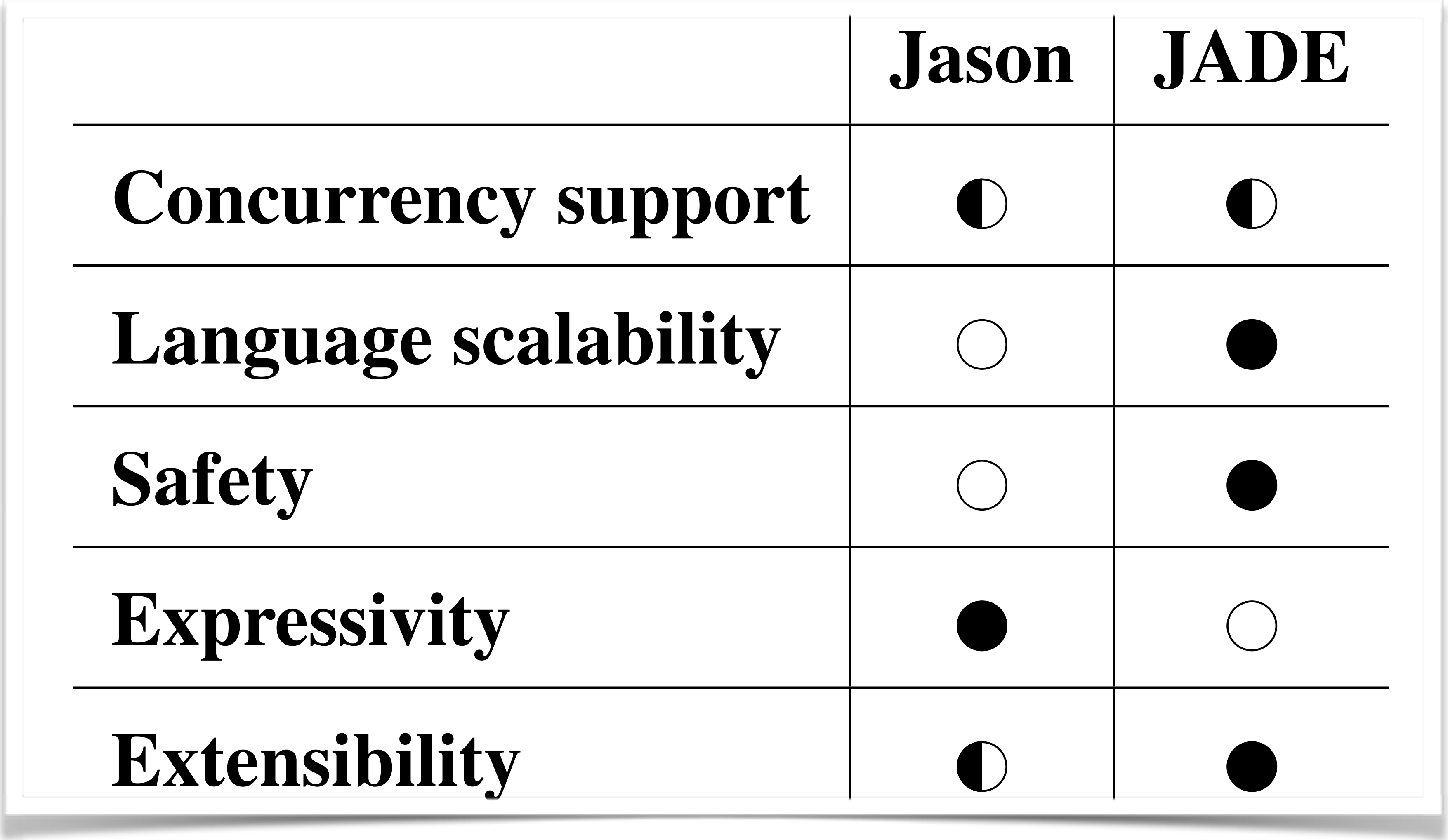
- slow interpreter
- not scalable

JADE's advantages

- manifestly typed
- scalable
- most popular agent framework

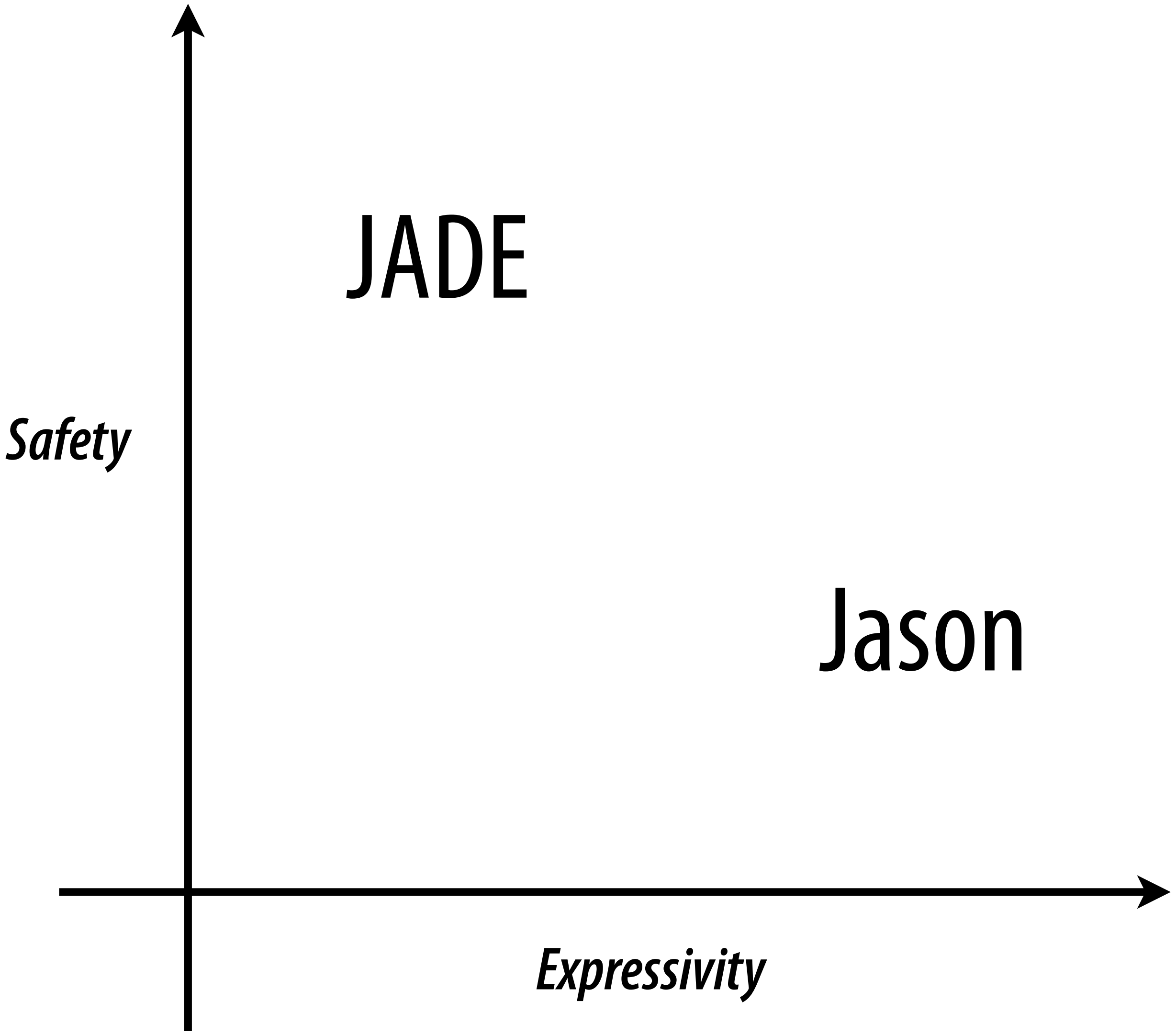
JADE's disadvantages

- lacks expressivity
- syntactic noise



	Jason	JADE
Concurrency support	◐	◐
Language scalability	○	●
Safety	○	●
Expressivity	●	○
Extensibility	◐	●

Comparing Jason and JADE

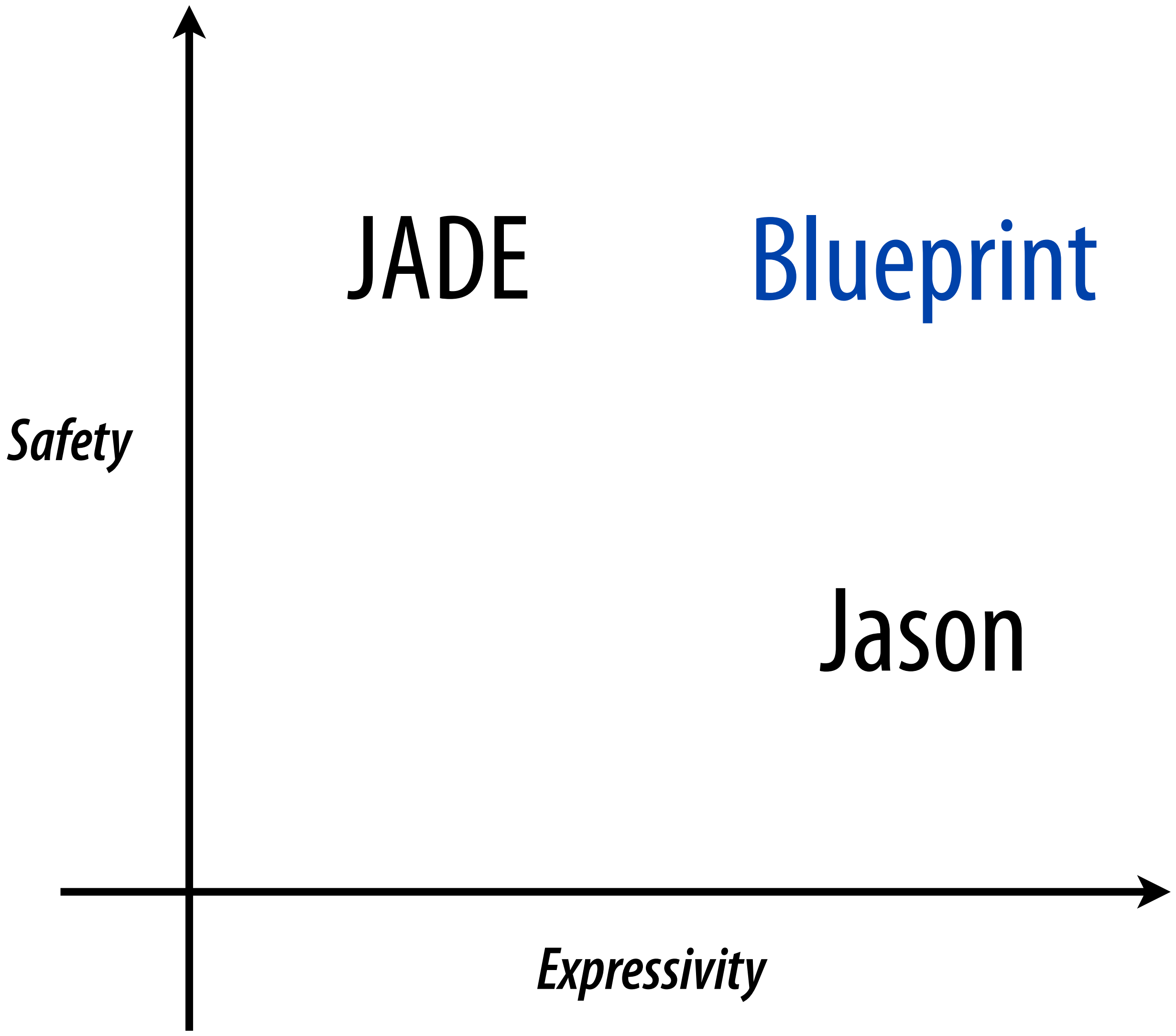


JADE

Safety

Jason

Expressivity



JADE

Blueprint

Safety

Jason

Expressivity

Monadic Foundations for Concurrent Agents

Monads

- originated in category theory
- structures that represent computation
- usually composed of a *type constructor* and *two operations*

Why monads?

- F#'s computation expressions are syntactic sugar for monads
- they are an elegant way of expressing the composition of concurrent computations
- they have been thoroughly studied

A closer look at concurrent computations

- they start *now* and they will finish sometime *in the future*
- we need to react when a concurrent computation ends
- we need to combine concurrently running computations

A closer look at concurrent computations

- the reaction to the completion of a concurrent computation is its *continuation*
- look at plans as being split in two: the *actions ran thus far* and the *actions that are still to be executed*
- a *promise* that a set of actions will get executed at some point
- this hints at a way of composing plans

The *Promise* monad

A promise for a value of type α is a function which receives a handler that can be called with the value of the promise, and it produces a value of type β. The *type constructor* for the Promise monad, $M_{promise}$, is defined as:

$$M_{promise} = (\alpha \to \beta) \to \beta$$

The *Promise* monad

The *unit operation* takes a value and returns a promise that will pass the value as an argument to the promise's handler:

$$unit_{promise} = \lambda x.\ \lambda k.\ k\ x$$

The *Promise* monad

The *bind operation* takes a promise and a continuation of the promise, and returns a promise that will invoke the continuation in a context where the result of the promise is available:

$$bind_{promise} = \lambda m.\, \lambda k.\, \lambda c.\, \boldsymbol{run}\; m\; (\lambda x.\, \boldsymbol{run}\; (k\; x)\; c)$$

where **run** is a function that executes a promise with the given callback

Conclusions

- the *Promise* monad is actually the well known *CPS* monad
- we can use monads to structure concurrent plans
- we can employ the same strategy as F#: use monads internally and perform code rewrite

The Blueprint Language

Blueprint is meant to be

- high level (e.g. agents, plans)
- safe (e.g. static types, channel protocols)
- easy to learn (e.g. C-like syntax)
- easy to use for concurrent applications

From revolution to evolution

- take a step back and look at agents as an evolution of the OOP and the Actors model
- *concurrently* executing agents with *reactive* behaviours
- respects Shoham's definition of AOP as a specialisation of OOP in the sense of the Actor model

The road to Blueprint

- agents are reactive and autonomous entities
- send messages asynchronously to mitigate deadlocks
- react to incoming events serially in order to avoid race conditions
- use monads to structure compose computations

Communication channels

- agents use *bidirectional* and *asymmetric* channels to exchange messages
- messages sends are asynchronous (i.e. non-blocking), while receives are synchronous (i.e. blocking)
- preserve message ordering
- they belong to exactly one agent
- they are introduced by the **chan** keyword

Channel endpoints

- an *exporting* endpoint, and an *importing* endpoint
- the exporting endpoint is used by the owner of the channel, while the importing endpoint can be handed off to other agents
- each endpoint has an *ordered*, *unbounded* message queue
- channel endpoints are first order entities (i.e. they can be passed as arguments and returned as values)

```
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    bel balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
```

```
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    bel balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
```

```
agent Account(init: int, impChan: BankAccount.Imp) {
    chan c = BankAccount.make()
    bel balance = init

    plan Start() {
        val msg = <-c.Exp.operation;
        match msg {
            case deposit(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance + amount)
            case withdraw(amount):
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
            case transferTo(acc, amount):
                acc <- deposit(amount);
                val currentBalance = balance.take()
                balance.put(currentBalance - amount)
        }
    }
}
```

Channel protocols

- declarative mechanisms of enforcing proper message exchange between agents
- specify the flow of the data between the communicating entities (i.e. the order, and direction in which messages are sent)
- introduced by the **`proto`** keyword

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

```
proto ThreadProto {
    start: in nextChan(next: ThreadProto.Imp@loop) >> loop
    loop: in token(value: Token) >> loop or end
}
```

Channel protocols

- protocols are designed from the perspective of the agent initiating the interaction (i.e. the exporting endpoint)
- there is no need to specify the dual protocol since it can be automatically derived by swapping direction specifiers

Concurrency and beliefs

- channels and protocols are a good way to control inter-agent concurrency
- we need a way to control intra-agent concurrency as well
- use *synchronised mutable variables* (mvars)

mvars

- *one-place buffers* which can be in one of the two states: *empty* or *full*
- two basic operations: *take*, and *put*
- calling take on a full mvar immediately returns the value and marks the mvar as empty
- If a take call is issued on an empty mvar, the calling thread of execution is blocked until the mvar becomes full
- the semantics of the put operations are similar

mvars

- the locks are not directly manipulated by the programmer, instead this is the job of the underlying implementation
- given the relatively low level, blocking nature of mvars (when compared to message passing), the risk of deadlock is still present

Beliefs as mvars

- Blueprint implements all beliefs as mvars
- beliefs are introduced by the **`bel`** keyword
- beliefs have two methods: take() and put()

Formal model sketch

- the semantics is defined via a CPS transform to a core language
- the core language is a small functional language

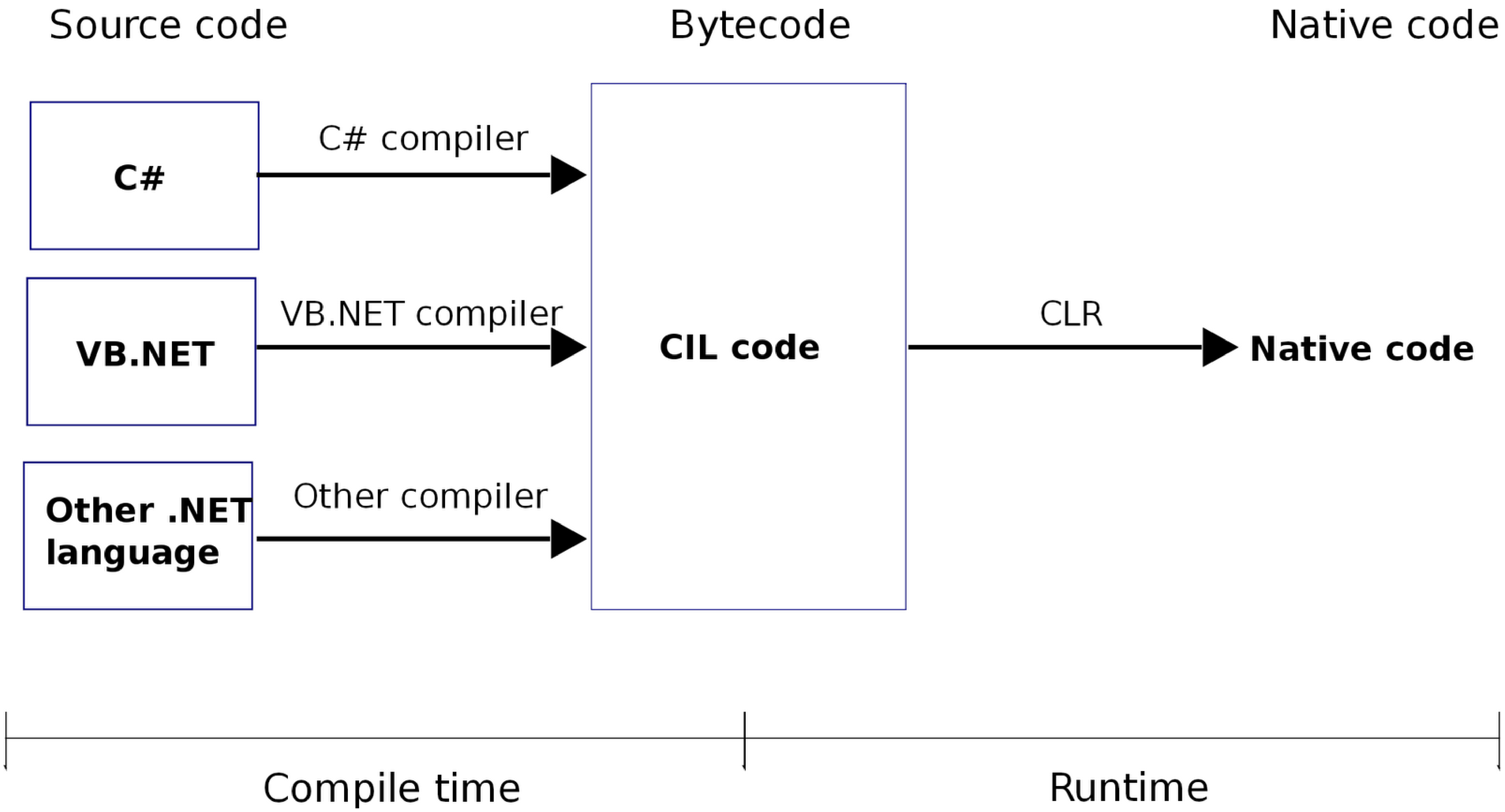
Proto π	::=	$\texttt{proto}\, id\ \{\sigma_0 \ldots \sigma_n\}$	Protocol definition
State σ	::=	$id : \overline{\mu} \gg \overline{id}$	Protocol state
MsgFlowExp $\overline{\mu}$	::=	$\mu_0 \rightarrow \ldots \rightarrow \mu_n$	Message flow expression
MsgExp μ	::=	$\texttt{in}\, id(id_0 \ldots id_n)$	Message receive expression
TargetStates $\overline{\sigma}$	::=	$id_0\ \texttt{or}\ \ldots\ \texttt{or}\ id_n$	Target states
	::=	$\texttt{out}\, id(id_0 \ldots id_n)$	Message send expression
Plan p	::=	$\texttt{plan}\, id(p_0 \ldots p_n)\ \{e\}$	Plan definition
Meth m	::=	$\texttt{def}\, id(p_0 \ldots p_n)\ \{e\}$	Method definition
Stmt s	::=	$\texttt{val}\, id = e$	Value binding
	::=	$\texttt{var}\, id = e$	Variable binding
	::=	$e_0; \ldots ; e_n$	Sequence
Exp e	::=	n	Numeral
		$true$	Boolean literal
		$false$	Boolean literal
		$''s''$	Literal
		id	Reference
		$e.id$	Field reference
		$e[i]$	Array element reference
		$e_1\ op\ e_2$	Binary operator
		$e(e_0 \ldots e_n)$	Function call
		$\leftarrow e$	Channel receive
		$e_1 \leftarrow e_2$	Channel send
		$e_1 := e_2$	Assignment
		ϵ	Empty expression

$$
\begin{aligned}
\llbracket \texttt{plan}\, id\, (p_0 \ldots p_n)\ \texttt{\{}\ e\ \texttt{\}} \rrbracket &\equiv \texttt{let}\, id = \lambda\, (p_0 \ldots p_n)\ .\ \lambda\, \kappa\ .\ \llbracket e \rrbracket \\
\llbracket e_{plan}() \rrbracket &\equiv \lambda\, \kappa\ .\ \texttt{\textbf{asyncstart}}\ (e); \kappa() \\
\llbracket e_1; e_2 \rrbracket &\equiv \lambda\, \kappa\ .\ \llbracket e_1 \rrbracket (\lambda\, ()\ .\ \llbracket e_2 \rrbracket\ \kappa) \\
\llbracket a \rrbracket &\equiv \lambda\, \kappa\ .\ a; \kappa() \\
\llbracket id := take(e) \rrbracket &\equiv \lambda\, \kappa\ .\ \texttt{\textbf{suspend}}\ (e, \lambda\, ()\ .\ set(id, take(e)); \kappa()) \\
\llbracket put(id, e) \rrbracket &\equiv \lambda\, \kappa\ .\ put(id, e); \texttt{\textbf{signal}}\ (id); \kappa() \\
\llbracket id := recv(e) \rrbracket &\equiv \lambda\, \kappa\ .\ \texttt{\textbf{suspend}}\ (id, \lambda\, ()\ .\ set(id, take(id)); \kappa())
\end{aligned}
$$

$$
\begin{array}{lrcll}
\textsc{Step}: & (\{e\} \cup A, Q, P) & \rightsquigarrow & (\{e'\} \cup A, Q, P) & , \ if \ e \mapsto e' \\
\textsc{Suspend}: & (\{ \ \texttt{\textbf{suspend}} \ (id, e)\} \cup A, Q, P) & \rightsquigarrow & (A, Q, P \cup \{id \rightarrow e\}) & \\
\textsc{Schedule}: & (A, \{e\} \cup Q, P) & \rightsquigarrow & (A \cup \{e\}, Q, P) & \\
\textsc{Signal}: & (\{ \ \texttt{\textbf{signal}} \ (id)\} \cup A, Q, P \cup \{id \rightarrow e\}) & \rightsquigarrow & (A, Q \cup \{e\}, P) & \\
\textsc{Async-Start}: & (\{ \ \texttt{\textbf{asyncstart}} \ (e)\} \cup A, Q, P) & \rightsquigarrow & (A, Q \cup \{e\}, P) &
\end{array}
$$

Implementation considerations

- Blueprint is built on top of the CLR framework
- The CLR contains a performant *Virtual Machine* with a *Just In-time Compiler* and a *Garbage Collector*
- we use the thread-pool pattern for scheduling agent reactions to incoming messages



Source code

Bytecode

Native code

C#

C# compiler

VB.NET

VB.NET compiler

Other .NET language

Other compiler

CIL code

CLR

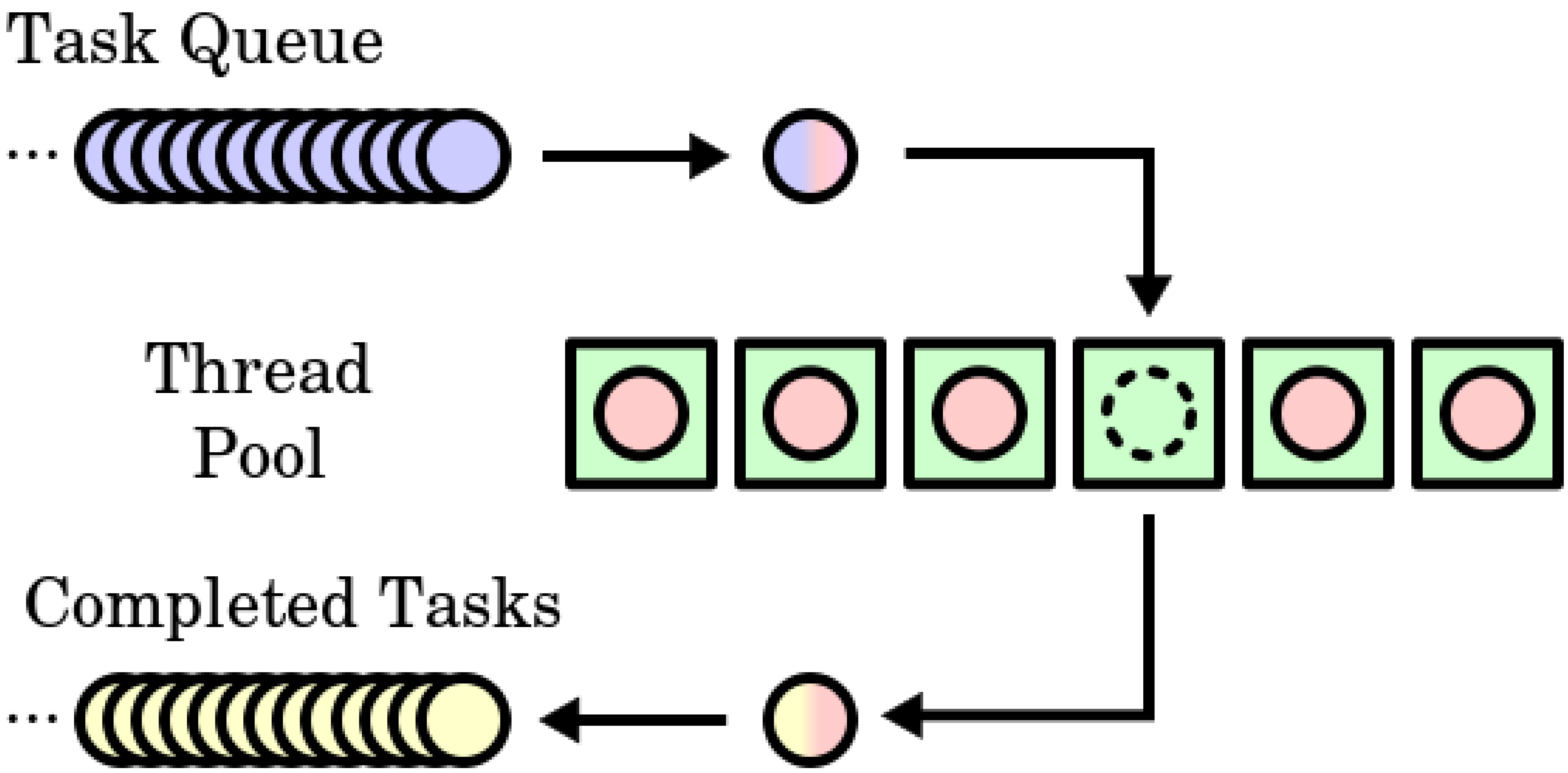
Native code

Compile time

Runtime

The thread-pool pattern

- a model where a (possibly fixed) number of threads—called worker threads—is created in order to execute waiting tasks—usually stored in a queue
- a worker thread requests the next pending task, and if one is available it runs it to completion
- the thread may sleep or it may request another task once the current task has finished


Task Queue
...
Thread
Pool
Completed Tasks
...

The thread-pool pattern

- it scales well for I/O-bound tasks
- the performance degrades when it has a lot of CPU-bound tasks

Future Directions

- investigate *code reuse* (most probably via some form of inheritance of prototypic delegation)
- investigate an extension of the concurrency model, based on the *Join calculus*
- give a full formal account of the language

- define a mechanism similar to channel protocols to characterise agent *environments*
- further investigate the object capability model in the context of security in AOP
- develop a JVM backend for Blueprint
- develop tooling for the language (i.e. plugins for popular IDEs)

Thank you. Questions?