The Syntax

1. Data ¶

The **Bach** language **[1]** , introduced following Linda, employs four key primitives for information manipulation:

- **tell**: places a piece of information on a shared space.
- **ask** : verifies the presence of a piece of information.
- **nask**: confirms the absence of a piece of information.
- **get** : checks the presence of information and removes one occurrence.

In the basic version, *BachT*, information comprises atomic tokens, forming a multiset in the shared space (store). While theoretically capable of coding various applications, this simplicity proves challenging in practical coding.

Consequently, more structured information emerges in the form of expressions like *f(a_1, … , a_n)*, where *f* is a functor and *a_1*, …, *a_n* are structured pieces of information. Notably, *tokens*, considered as basic pieces of information, are devoid of arguments and are represented without parentheses.

Example: In the Needham-Schroeder protocol, Alice and Bob use the tokens na and nb to represent their respective nonces. Additionally, their public keys are coded using the tokens *pka* and *pkb*. An encoded message, produced by Alice encrypting her nonce along with her identity using Bob's public key, is represented as the following structured piece of information: *encrypt(na, alice, pkb)*..

The set of structured pieces of information is abbreviated as *{cal I}*, and the term "si-term" is later employed to refer to a structured piece of information.

2. Agents

In **Bach**, primitives like *tell*, *ask*, *nask*, and *get* manipulate a shared space, formally defined as a multiset of structured information. Transition rules govern their execution.

- Composition operators (;, ||, +) combine primitives to create complex agents.
- Procedures, defined with *proc*, associate an agent with a name, following the guardedness principle.

Example: The behaviors of agents, such as Alice and Bob, can be coded using these primitives and operators.

```
proc Alice = tell(encrypt(na,a,pkb));  get(encrypt(na,nb,pka));   tell(encrypt(nb,pkb)).

     Bob   = get(encrypt(na,a,pkb);    tell(encrypt(na,nb,pka));  get(encrypt(nb,pkb)).
```

3. The Scala Programming Language

Scala [2], a statically typed language, seamlessly integrates object-oriented and functional programming. It ensures code safety through static typing, with features like type inference enhancing readability. Scala's functional programming support. Immutability, and pattern matching make it suitable for scalable applications.

Variables in Scala can be either immutable or mutable, as shown below:

```
val immutableVariable: Int    = 42

var mutableVariable  : String = "Hello, Scala!"
```

Methods, introduced with the *def* keyword, can be generic, written in curried form, and specify return types:

```
def add(x: Int, y: Int): Int = x + y
```

In Scala, methods are typically part of object, class, or trait definitions. *Case classes* automatically define setter, getter, hash, and equal methods. Notable features of Scala include.

3.1. Functions and objects:

Functions can be coded using objects with an apply function.

Example:

```
object tell  { def apply(siterm: SI_Term)  = TellAgent(siterm)  }

object Agent { def apply(agent: BSC_Agent) = CalledAgent(agent) }
```

3.2. Strictness and laziness:

Scala is a strict language but supports call-by-name and thunks for delayed evaluation. The call-by-name strategy is exemplified by:

Example:

```
def doubleFirstLazy(x: Int, y: => Int) = x + x
```

Thunks, functions with no arguments, delay evaluation until needed. The *lazy* keyword can also be used to defer the evaluation of val-declared expressions:

```
lazy val recursiveExpression =  ... recursiveExpression ...
```

4. Constrained executions

The interpretation of Bach agents in the B2Scala tool provides the ability to selectively choose computations of interest. This is achieved by expressing logical formulae that need to be satisfied. The logic employed is inspired by the Hennessy-Milner logic **[3]** and the π-calculus **[4]**. Specifically tailored for coordination scenarios, it relies on basic formulae asserting the presence or absence of si-terms.

Example: *bf(i_running(Alice,Bob))* affirms that Alice and Bob have initiated a session. These formulae can be combined using classical logical operators like and, or, and negation, denoted as bf-formulae.

Similar to Hennessy-Milner logic, bHM-formulae are employed to specify sequences of properties that must hold across the sequences of stores generated by computations. They are also designed to allow for choices of paths and are typically denoted as *h*, *h_1*, *h_2*. They are inductively defined with the following grammar:

```
bHM  ::=  f  |  P  |  h_1 + h_2  |  h_1 ; h_2
```

Here, *f* represents a bf-formula, *h_1* and *h_2* are bHM-formulae, and *P* is a variable defined by an equation of the form *P* = *h*. Similar to agents, it is assumed that *h* is guarded, meaning that a bf-formula is required before the variable *P* is called recursively.

As an example, an attack on the Needham-Schroeder protocol can be identified by finding a computation satisfying the bHM-formula *X* defined as:

```
  X   =  (  not(i_running(Alice,Bob)) ;  X  ) + r_commit(Alice,Bob)
```

Footnotes

[1] Jacquet, J.M., Barkallah, M.: Anemone: A workbench for the Multi-Bach Coordination Language. Science of Computer Programming 202, 102579 (2021)

[2] M. Odersky, L. Spoon, B. Venners, Programming in Scala, A comprehensive step-by-step guide, Artemis, 2016.

[3] Hennessy, M., Milner, R.: On Observing Nondeterminism and Concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) Proceedings of the International Conference on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 85, p. 299–309. Springer (1980)

[4] Kozen, D.: Results on the Propositional mu-Calculus. Theoretical Computer Science 27, 333–354 (1983)