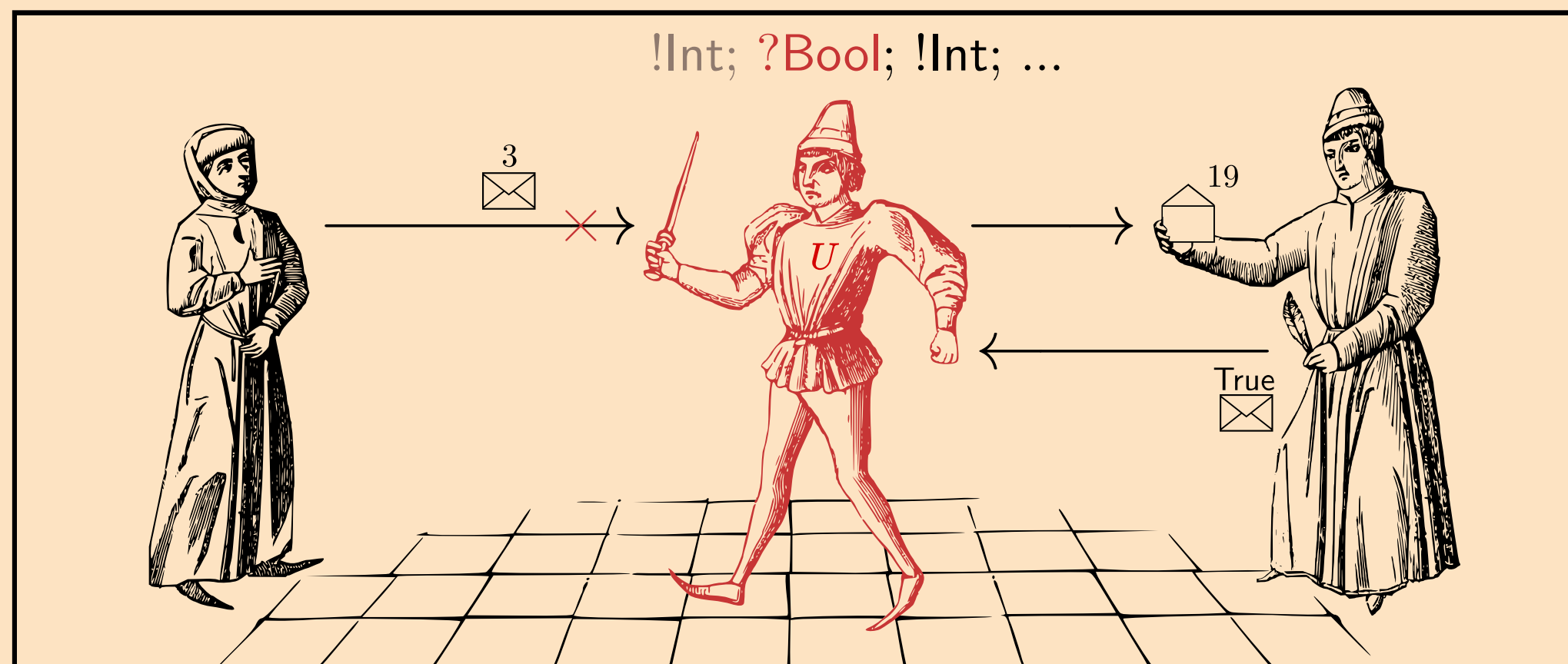


Subtyping Context-Free Session Types



SESSION types allow the specification of structured communication protocols on bidirectional, heterogeneously typed channels. Typically, these specifications include the type, direction and order of messages, as well as branching points where participants can decide how communication should proceed. Context-free session types enhance the expressivity of regular session types by enabling the specification of a wider class of protocols, those corresponding to (simple and deterministic) context-free languages.

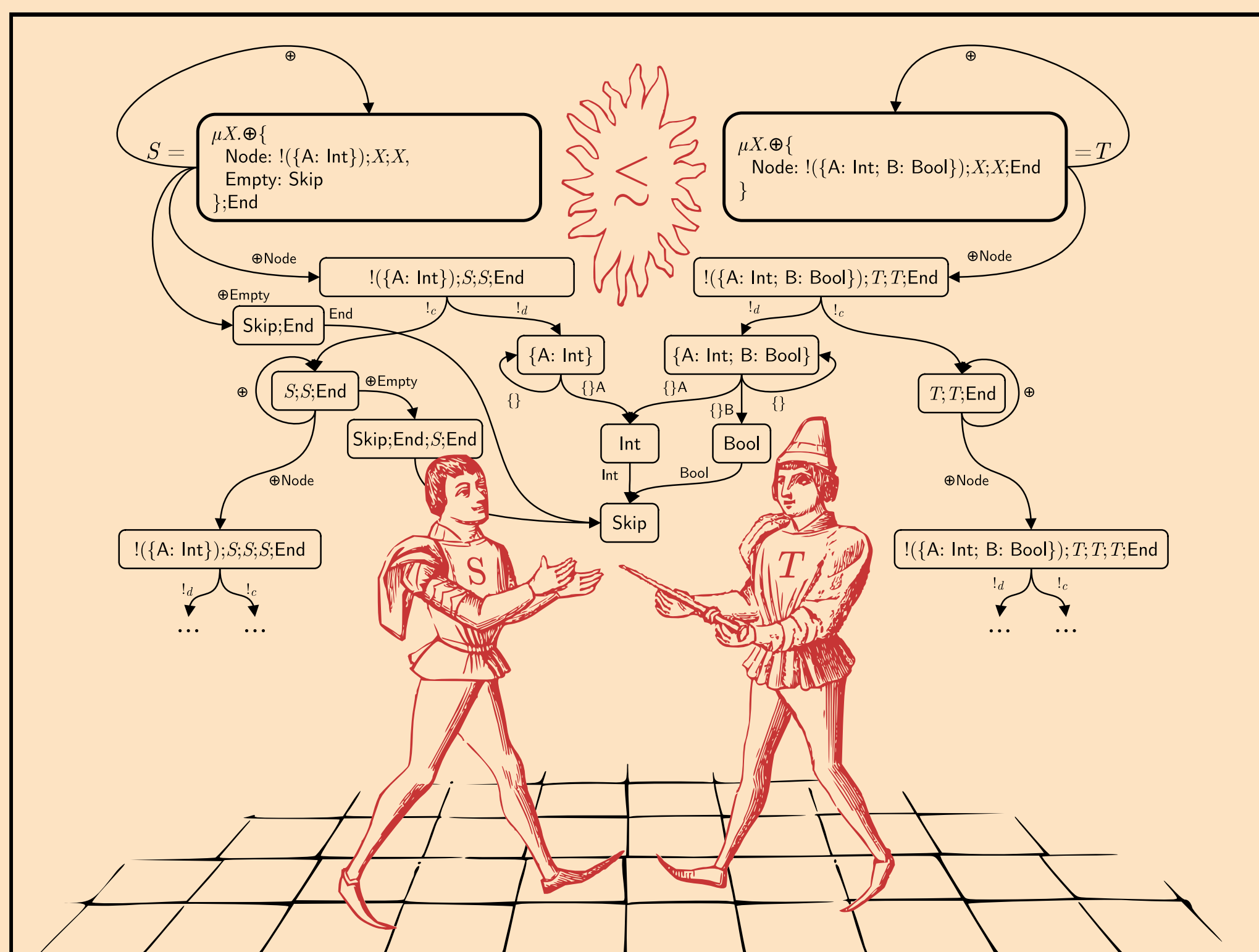
Support for session types in a programming language provides for safer concurrent programming by allowing its typechecker to validate the behavior of programs against these types, ensuring session fidelity, privacy, communication safety and, in some variants, deadlock freedom.



Subtyping for session types allows increased flexibility in the interactions between session participants, namely on the type of messages they exchange and on the choices each participant has available at a branching points. A practical benefit of this flexibility is that it promotes modular development: the behaviour of one participant (e.g., a server) may be refined, while that of the other (e.g., a client) is kept intact.


While the algorithmic properties of subtyping for regular session types are by now well known, the same cannot be said for their context-free counterparts, which exhibit complex algebraic properties. A previous investigation into this topic has shown this problem to be undecidable – the usual, unfortunate price to pay for expressive power.

Despite these challenges, we present a semantic approach to subtyping for context-free session types, supported by a novel kind of observational preorder we call XYZW-simulation, which allows us to selectively combine the requirements of simulation, reverse simulation and contra-simulation in order to handle the covariant and contravariant properties of session type constructors to their full extent.



XYZW-similarity generalizes bisimilarity, on which the notion of type equivalence for context-free session types is based. Taking advantage of this fact, we are able to derive a sound (but necessarily incomplete) subtyping algorithm from an existing type equivalence algorithm based on tree search.

We implemented our algorithm in the FreeST programming language compiler, which natively supports context-free session types. In order to evaluate its performance, we employed unit tests, generative testing using Quickcheck and program tests drawn from the original FreeST test suite.

FreeST A statically typed concurrent functional language with context-free session types 

An example of an inherently context-free protocol is the serialization of JSON data on a single channel. Should a general JSON deserializer be able to receive data from a number array serializer? Of course. But this is only possible thanks to subtyping!

```
data Json = Null | Boolean Bool | Number Int | String String
          | Array [Json]
          | Object [(String, Json)]

type DeserializeJson = &{
  Object : RcvObject,
  Array : RcvArray,
  String : !String,
  Number : !Int,
  Boolean: !Bool,
  Null   : Skip
}

type SerializeNumberArray = +{
  Array : SerializeNumbers
}

type SerializeNumbers = +{
  ANil : Skip,
  ACons: +{Number: !Int};SerializeNumbers
}

type DeserializeArray = &{
  ANil : Skip,
  ACons: DeserializeJson;DeserializeArray
}

(...)

serializeIntList : [Int] -> SerializeNumberArray; a -> a
serializeIntList l c = c |> select Array |> sendInts l

sendInts : [Int] -> SerializeNumbers; a -> a
sendInts [] c = c |> select ANil
sendInts (n::ns) c = c |> select ACons |> select Number |> send n |> sendInts ns

main : Json
main =
  let (r,s) = new @(RcvJson;End) () in
  fork (\_ 1-> s |> serializeIntList [1,2,3,4] |> close);
  let (j, c) = deserializeJson r in close c;
  j
```