Making Session Types Go

Compilation of a session-typed functional language to Go

João Geraldo[1] and Bernardo Toninho[1]

FCT NOVA and NOVA-LINCS
j.geraldo@campus.fct.unl.pt
btoninho@fct.unl.pt

Abstract. Session types are a typing discipline for message-passing concurrency that is able to ensure strong compile-time correctness guarantees for concurrent programs. In this work we develop a compiler for a session-typed functional language targeting the Go language. Our work features standard functional programming features combined with channel-based, session-typed, concurrency primitives and thread spawning. Concurrency and pure functional values are separated via a monad-like interface, in the style of Haskell. Our compilation pipeline takes a program, type checks it to ensure the absence of deadlocks and communication errors and then translates it to valid Go code, leveraging Go's channel and lightweight thread infrastructure. The translation requires compensating the mismatch between Go's channel types and session types, which we achieve via a state machine view of session types.

Keywords: Session Types · Compilation · Go.

1 Introduction

Channel-based concurrency is a form of concurrent programming that aims to alleviate the intricacies of shared memory concurrency by having concurrent threads or processes exchange data via communication rather than interference over shared data structures. Typically, this form of concurrency does not require the use of low-level concurrency control mechanisms such as locks and provides easier to use, higher-level concurrency primitives. Modern programming languages such as Go, Rust or Haskell provide support for channel-based concurrency, with limited compile-time correctness guarantees.

In these languages, channels are statically typed to carry payloads of a given *fixed* type, and a well-typed program is only ensured to never attempt to communicate over a channel using values of the wrong type. Since most programs need to exchange values of many different types, concurrent programs must make *coordinated* use of many different channels. This problem is aggravated by the fact that these channels are often unidirectional, further adding to the coordination needs and giving rise to programming errors that can result in deadlocks.

This is in sharp contrast with the state of the art typing disciplines for channel-based concurrency, which often ensure at compile time both communica-

tion safety (absence of communication mismatches) and deadlock-freedom. Session types [12] are one such typing discipline, where channels are typed according to simple *communication protocols*. For instance, the session type $int \wedge string \supset \mathbf{1}$ (using the syntax of Section 2.2) specifies a channel along which we send exactly one value of type *int*, afterwards input a *string* and then stop using the channel. In a language with session types, type checking enforces that the channel is used according to this specification, effectively eliminating communication mismatch errors. In session type systems based on linear logic [26] (such as those this work is based on), typing further ensures that no deadlocks occur.

Session types are generally absent from general purpose languages since they require so-called *linear* typing, in order to track the stateful nature of a channel's type, where the valid payload type of a channel at a given moment depends on the previous actions taken on the channel. This results in implementations realized as DSLs or libraries (e.g. [13,27]) which provide diminished compile-time guarantees; or in special purpose languages, designed specifically with session types in mind (e.g. [1]), but lacking many of the modern features and ecosystems of fully-fledged general purpose languages. Performance is usually a further issue, since these languages are often *interpreted* rather than *compiled*.

In this work we introduce a session-typed functional language, following [26], which provides compile-time guarantees of absence of communication errors and deadlocks. Unlike other natively session typed languages, we develop a *compiler* from our language to valid Go code, fully taking advantage of Go's channels and lightweight threads. Thus, we can provide the compile-time correctness assurances of session typing, while alleviating many of the limitations of having a special purpose language.

Paper structure. We introduce the language and its typing in Section 2. The challenges of type checking and compiling our language to Go are described in Section 3 and we showcase the expressiveness of our language via examples (Section 4). We conclude with a discussion of related and future work (Section 5).

2 Language

Our language consists of a session-typed functional language in the style of the SILL family [26,25,21] of languages based on the propositions-as-types interpretation of intuitionistic linear logic as a session-typed language [2]. The language features channel-based concurrency combined with standard functional programming features. In our language, concurrency primitives form a sub-language which we dub the *process* layer, and our typing discipline enforces a strict separation between the standard functional connectives (the *functional* layer) and the process layer, reminiscent of monadic programming in Haskell [28].

Terms in the process layer, often referred to as *processes*, may freely use terms from the functional layer (e.g. it is possible to communicate functional values). The functional layer, on the other hand, can only depend on process terms in a controlled way, so as to enforce the strong static correctness of the

$$
\begin{array}{rcl}
M, N & ::= & V \mid \texttt{fun}\ \overline{x_i} \to M\ \texttt{end} \mid \texttt{let}\ x = M\ \texttt{in}\ N \mid M\ N \\
& \mid & \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2\ \texttt{endif} \mid c \leftarrow \{P\} \leftarrow \overline{d} \\
P, Q & ::= & \texttt{send}\ c\ M\ ; P \mid \texttt{send}\ c\ d\ ; P \mid x : T \leftarrow\ \texttt{recv}\ c\ ; P \mid \texttt{close}\ c \\
& \mid & \texttt{wait}\ c; P \mid \texttt{fwd}\ d\ c \mid c \leftarrow \texttt{spawn}\ M\ \overline{d}; P \mid \texttt{case}\ c\ \texttt{of}\ \overline{l_j : (P_j)} \\
& \mid & c.l; P \mid \texttt{if}\ M\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{endif} \mid \texttt{print}\ M; P
\end{array}
$$

Fig. 1. Syntax of Expressions (M, N) and Processes (P, Q)

overall language. As we detail below, processes are *embedded* in the functional layer, becoming possible to define processes via functions.

2.1 Syntax

The syntax of our language is given in Figure 1. We range over functional terms with M, N and over process terms with P, Q. A program in our language is a sequence of declarations of the form $\texttt{let}\ x : T = N$, where the name x (of type T) can occur in N, in order to allow for recursive (function) definitions. We distinguish the top-level declaration of name `main`, which acts as the program's entry point. We also allow for (session) type declarations.

Functional terms form a standard functional language with the expected constructs: multi-argument λ-abstractions take the form $\texttt{fun}\ \overline{x} \to M\ \texttt{end}$ (we often omit type annotations in λ-bound variables due to our use of bidirectional type-checking – Section 3.1); function application is noted as $M\ N$; potentially recursive let-bindings are written $\texttt{let}\ x = M\ \texttt{in}\ N$; basic values (e.g. numbers, strings, booleans) and their operators (e.g. arithmetic, relational operators, etc.), are abstracted by the meta-variable V. We highlight the construct $c \leftarrow \{P\} \leftarrow \overline{d}$ which internalizes a process term P as an *opaque* functional value (i.e., no evaluation of P takes place), where the process P *offers* some session behavior on channel c, while *using* channels $\overline{d}$ (both c and $\overline{d}$ are bound in P).

The process constructs codify the session behavior usage on the appropriate channels. For instance, $\texttt{send}\ c\ M\ ; P$ denotes a send action on channel c of term M (which will be fully evaluated before communication), with continuation behavior given by process P. Dually, $x : T \leftarrow\ \texttt{recv}\ c\ ; P$ performs an input action on channel c, binding the received value (of type T) to x in the continuation P. For the sake of conciseness we overload send and receive operations, using similar syntax for communication of functional data and for (higher-order) communication of channels. The construct $\texttt{close}\ c$ signals that no further communication will take place on channel c, whereas construct $\texttt{wait}\ c; P$ waits for the closure of session channel c. The process construct $\texttt{fwd}\ d\ c$ establishes a forwarder between channels c and d (which must be of the same type), essentially redirecting inputs and outputs on c to d and vice-versa.

As is usual in session-based communication, we allow selection and branching constructs: the construct $\texttt{case}\ c\ \texttt{of}\ \overline{l_j : (P_j)}$ denotes an *external* choice on channel c, and so the corresponding process will receive on c some label l_i and proceed according to the continuation process P_i (typing ensures that all possible

branching options are covered); dually, the construct $c.l; P$ signals on channel c the selection of the branch labelled by l, with continuation P.

Finally, we note the construct $c \leftarrow \texttt{spawn}\ M\ \overline{d}; P$, which is the process-level dual of the process embedding (functional) construct $c \leftarrow \{P\} \leftarrow \overline{d}$. While the latter embeds processes as values in the functional layer, the former allows for the *usage* of such values in other processes. Specifically, the execution of process $c \leftarrow \texttt{spawn}\ M\ \overline{e}; P$ will eventually evaluate M to an embedded process of the form $c \leftarrow \{Q\} \leftarrow \overline{d}$ (guaranteed by type safety), where channel c is bound in the continuation P and $\overline{e}$ denotes a list of (free) channels that will instantiate the channels $\overline{d}$ in Q. Process P and Q will subsequently execute in parallel, sharing the channel c for inter-process communication.

The example below showcases a program combining several features of our language. First, we define a *recursive* session type dubbed `IntStream`, which codifies the behavior of emitting an infinite stream of integers. Function `nats` takes an integer `n` and produces a process offering such an `IntStream` on channel `c`. The process sends the number `n` on `c` and afterwards *spawns* a recursive call of `nats (n+1)` on channel `d`, which is then connected to the offering channel `c` via the forwarding construct `fwd` d c. Thus, we can define recursive *processes* via a combination of recursive *functions*, spawn and forwarding:

```
stype IntStream = rec x. int ^ x;
let nats : int -> {IntStream} =
fun n -> c <- { send c n;
                d <- spawn (nats (n+1));
                fwd d c } end;

let main : {IntStream} = (nats 0);
```

The example above further showcases how process expressions in our language may execute. While embedded processes are values, we treat the function `main`, which must have a process value type, as the top-level function which may execute a process (similar to how `main` in Haskell is the function that actually triggers the execution of effectful computation).

The following examples illustrate the use of choice in our language:

```
stype IntCStream = rec x.&{next: int^x, stop: 1};
let augNats : int -> {IntCStream} =
fun n ->  c <- {
          case c of
          next: send c n;
                d <- spawn (augNats (n+1));
                fwd d c
          stop: close c } end
```

```
c <- {  d <- spawn {
                d.l1
                send d (5*3);
                close d };
case d of
 l1: v <- recv d;
     wait d;
     close c
 l2: send d 10;
     send d 100;
     wait d;
     close c     }
```

The example on the left shows how the previous `nats` function can be augmented from an infinite stream of integers to one that can be potentially finite. The process that repeatedly communicates along `c` now offers a choice of two behaviors: `next` and `stop`. If `next` is received on `c`, the process will behave as `nats` and the stream will emit its next number. If `stop` is received, the process will close its channel and terminate. The example on the right shows both selection and branching by spawning a process which emits label `l1` to identify its

$$
\begin{array}{lll}
D & ::= & \texttt{stype}\ N = A \\
T, S & ::= & \mathsf{Num} \mid \mathsf{Bool} \mid \cdots \mid T \to S \mid \{\overline{B} \vdash A\} \\
A, B & ::= & A \multimap B \mid A \otimes B \mid T \wedge B \mid T \supset B \mid \&\{\overline{l_i : A_i}\} \mid \oplus\{\overline{l_i : A_i}\} \mid \mu X.A \mid X \mid N \mid \mathbf{1}
\end{array}
$$

Fig. 2. Functional Types (T, S), Session Types (A, B) and Declarations (D)

subsequent behavior, in parallel with a process that inputs either `l1` or `l2`, with the `l1` branch matching the (dual) behavior of the spawned process.

2.2 Typing

The syntax of types for our language is given in Figure 2, following [26]. We distinguish between functional types, ranged over by T, S, which type functional terms, and session types A, B which type *channels* used in process terms. Our language also supports session type declarations (D) for convenience. Functional types are mostly standard, consisting of basic data types such as numeric types and booleans and function types $T \to S$. We note the type $\{\overline{B} \vdash A\}$, which types a functional term of the form $c \leftarrow \{P\} \leftarrow \overline{d}$, where process P will *offer* session type A on channel c, using channels $\overline{d}$ according to types $\overline{B}$.

Session types characterize sequences of communication actions that take place on channels: the type $A \multimap B$ denotes a channel on which its offering process expects to *receive a channel* typed with A to then behave according to the session type B; dually, type $A \otimes B$ denotes a channel on which its offering process will send a channel of type A and then behave according to type B; types $T \supset B$ (concrete syntax T `=>` B) and $T \wedge B$ (concrete syntax T `^` B) denote input and output of functional values of type T with continuation type B, respectively; type $\&\{\overline{l_i : A_i}\}$ denotes a channel along which its offering process will receive one of the labels l_i and then offer the behavior prescribed by session type A_i; dually, session type $\oplus\{\overline{l_i : A_i}\}$ denotes a channel along which one sends one of the labels l_i to then behave according to A_i; type $\mathbf{1}$ denotes the inactive session; and type $\mu X.A$ and type variables X allow for recursive session types. Note that named types N may be used accordingly.

Our language features two typing judgments, written $\Psi \Vdash M : T$ and $\Psi; \Delta \vdash P :: c{:}A$. The former states that functional term M has type T under the typing assumptions of the form $x{:}T$ for free variables, tracked in context Ψ. The latter states that process P offers *session type* A along channel c by using the session behaviors specified in the *linear* typing context Δ and under the (functional) typing assumptions Ψ.

We omit the full set of typing rules, emphasizing the key rules. We begin with the typing rule for the construct that embeds processes in functional values:

$$
\frac{\Psi; \overline{d : B} \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P\} \leftarrow \overline{d} : \{\overline{B} \vdash A\}}\ (\{\}\text{-I})
$$

The rule above states that the *functional* term $c \leftarrow \{P\} \leftarrow \overline{d}$ has type $\{\overline{B} \vdash A\}$ provided its underlying process P is well-typed in a context where it will *use* channels $\overline{d}$ with types $\overline{B}$ and offer type A along channel c. Dually, we may use such values within other processes as codified by the following rule:

$$\frac{\Psi \Vdash M : \{\overline{B} \vdash A\} \qquad \Delta' = \overline{d : B} \qquad \Psi; \Delta, c{:}A \vdash P :: f{:}C}{\Psi; \Delta, \Delta' \vdash c \leftarrow \texttt{spawn}\ M\ \overline{d}; P :: f{:}C}\ (\{\}\text{-E})$$

The rule above captures the spawning of the (process) value M in parallel with process P by first typing M as a process value of type $\{\overline{B} \vdash A\}$. Since such a process must be provided with sessions $\overline{B}$, spawning it requires satisfying this constraint with some available channels $\overline{d}$, which will be consumed by the process. This is captured by isolating the context region Δ' which satisfies this constraint and is no longer available for use by process P. Having satisfied the constraints required to run M, the process P is then warranted in interacting with such a process via the channel c, which will be offered at type A.

The typing rules for process terms codify how to *use* and *offer* sessions at a given type. The offered session is available on the distinguished right-hand side channel, which we often write as c. The used channels are tracked in the context Δ. As a kind of special case we have the *forwarding* construct:

$$\frac{}{\Psi; d{:}A \vdash \texttt{fwd}\ d\ c :: c{:}A}\ (\textsc{fwd})$$

Given a single ambient d channel of a given type A, we can *use* it to offer a behavior of the same type along c by essentially forwarding all messages between the two channel endpoints (e.g. messages sent along d are redirected to c and vice-versa – see Section 3.3 for more on how the forwarding behavior is implemented).

Value communication is typed by the following rules:

$$\frac{\Psi \Vdash M : T \qquad \Psi; \Delta \vdash P :: c{:}A}{\Psi; \Delta \vdash \texttt{send}\ c\ M\ ; P :: c{:}T \wedge A}\ (\wedge\text{-R}) \qquad \frac{\Psi, x{:}T; \Delta, c{:}A \vdash P :: d{:}D}{\Psi; \Delta, c{:}T \wedge A \vdash x \leftarrow\ \texttt{recv}\ c\ ; P :: d{:}D}\ (\wedge\text{-L})$$

Rule ($\wedge$-R) allows us to type a process term that *offers* a session of type $T \wedge A$ by sending a term M of type T along channel c and then offering the behavior A along c. Dually, to *use* such a session we must *receive* a value of type T, bound to x in the continuation process P, which may then subsequently use the channel as type A. The remaining typing rules for process terms follow a similar pattern, the so-called *right* rules (marked with R) codify typing of processes offering sessions, and *left* rules (marked with L) codify typing of processes using ambient sessions of a given type.

(&-R)
$$\frac{\Psi; \Delta \vdash P_1 :: c{:}A_1 \qquad \ldots \qquad \Psi; \Delta \vdash P_n :: c{:}A_n}{\Psi; \Delta \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j : (P_j)} :: c{:}\,\&\,\{\overline{l_j : A_j}\}}$$

(&-L)
$$\frac{\Psi; \Delta, c{:}A_i \vdash P :: d{:}D}{\Psi; \Delta, c{:}\,\&\,\{\overline{l_j : A_j}\} \vdash c.l_i; P :: d{:}D}$$

We show the rules for the choice type $\&\{\overline{l_j : A_j}\}$ above, where a process offers a session of type $\&\{\overline{l_j : A_j}\}$ by providing with an alternative l_i, for each of the choice labels, with the appropriate type A_i, on channel c. Using such a session requires commiting to one of the possible choices l_i, which is sent over channel c and then warrants the use of c as a session of type A_i.

Typing ensures that no well-typed program can result in a deadlock, as well as the absence of communication mismatch errors [26]. While all examples presented so far are well-typed (i.e., deadlock-free), processes `P1` and `P2` below

```
let P1 = c <- { send c 1 ;                  let P2 = d <- { send d 2 ;
                x <- recv d ;                               x <- recv c ;
                wait d ;                                    wait c;
                close c } <- d                              close d } <- c
```

are individually well-typed, but the parallel composition of the two, which deadlocks due to the wrong ordering of actions on channels `c` and `d`, is ill typed. In fact, our type-based analysis conservatively excludes cycles in the topology of communicating processes.

3 Implementation

Our implementation (in OCaml) is split in two stages: *type checking*, which consists of an implementation of the typing system introduced in Section 2.2, using the technique of *bidirectional* type-checking [22,9]; and *compilation* to executable Go code. The type checking process further augments the abstract syntax tree with reconstructed typing information that is omitted in the user-level syntax, to be used in the compilation stage. Compilation is split in two steps: a *preamble generation* step, where the session types in a program are converted into a set of types and methods in Go; and the code generation step, where the instructions of the program are compiled into Go code.

3.1 Type Checking

We implement the type checker for our language by making use of the technique of bidirectional typing to reformulate the declarative typing judgments $\Psi \Vdash M : T$ and $\Psi; \Delta \vdash P :: c{:}A$ in terms of algorithmic *checking* and *synthesis* judgments.

For the functional layer, the approach is standard and so we omit the rules for the new judgments: we consider algorithmic judgments $\Psi \Vdash M \Rightarrow T$ and $\Psi \Vdash M \Leftarrow T$, which denote that M synthesizes type T and M checks against type T, respectively. Each rule in the declarative system corresponds to either a synthesis or a checking rule. The main practical consequence of this approach is that we need only have type annotations in top-level definitions and can omit type annotations in all other binders entirely.

For the process layer, the algorithmic formulation of the typing rules must also account for the linear treatment of the context Δ, where *all* ambient sessions must be fully used. We achieve this following the approach of [3], formulating the algorithmic version of the system in terms of *input* and *output* channel

contexts: $\Psi; \Delta_I / \Delta_O \vdash P \Rightarrow c{:}A$ and $\Psi; \Delta_I / \Delta_O \vdash P \Leftarrow c{:}A$. The input context Δ_I identifies the channels that are necessarily used by P. The output context Δ_O tracks leftover channels that must still be used. The use of $\Leftarrow$ and $\Rightarrow$ to denote synthesis and checking is as in the functional setting. For instance, the algorithmic formulation of the typing rule for process composition is:

$$\frac{\Psi \Vdash M \Rightarrow \{\overline{B} \vdash A\} \qquad \Delta_I' = \Delta_I \setminus \{\overline{d{:}B}\} \qquad \Psi; \Delta_I', c{:}A / \Delta_O \vdash P \Rightarrow f{:}C}{\Psi; \Delta_I / \Delta_O \vdash c \leftarrow \texttt{spawn}\ M\ \overline{d}; P \Rightarrow f{:}C}\ (\{\}\text{-E})$$

We can synthesize the type for $c \leftarrow \texttt{spawn}\ M\ \overline{d}; P$ under input context Δ_I by first synthesizing the type of M, which allows us to calculate the input context for the typing of P from the input context Δ_I by removing from it the channels $\overline{d{:}B}$ and adding a new binding $c{:}A$. The output context for the typing of spawn is propagated from the output context for the typing of P.

The checking and synthesis modes are mediated by the following rules:

$$\frac{\Psi; \Delta_I / \Delta_O \vdash P \Rightarrow c{:}A \quad A \leq B}{\Psi; \Delta_I / \Delta_O \vdash P \Leftarrow c{:}B}\ (\textsc{sub}) \qquad \frac{\Psi; \Delta_I / \Delta_O \vdash P \Leftarrow c{:}A}{\Psi; \Delta_I / \Delta_O \vdash (P :: c{:}A) \Rightarrow c{:}A}\ (\textsc{annot})$$

The (SUB) rule incorporates session subtyping $A \leq B$ into the algorithmic system [11]. In our language, subtyping is used to handle unfolding and folding of recursive session types. For instance, through the (SUB) rule, a process offering a session of type $\mu X.\&\{\mathsf{next} : \mathit{int} \wedge (\mu X.\&\{\mathsf{next} : \mathit{int} \wedge X, \mathsf{stop} : \mathbf{1}\}), \mathsf{stop} : \mathbf{1}\}$ can be treated as one offering a session of the folded type $\mu X.\&\{\mathsf{next} : \mathit{int} \wedge X, \mathsf{stop} : \mathbf{1}\}$, without the need for explicit fold or unfold annotations in the syntax.

3.2 Compilation

We chose to compile our language to Go in order to take advantage of Go's efficient, channel-based concurrency primitives and lightweight threads (*goroutines*), instead of having to deploy a likely less efficient concurrent runtime. We note that the Go compiler does not easily support compiler extensions nor does Go have a reasonable DSL ecosystem that would enable us to embed our language as an eDSL in Go. Moreover, by compiling to a high-level language that is similar to our source language we can leverage the compiler optimizations of the Go compiler rather than having to deploy these techniques from scratch.

The compilation of functions and processes is mostly straightforward. Functions are compiled directly to Go functions. A process value of the form $c \leftarrow \{P\} \leftarrow \overline{d}$ is compiled into a Go function which takes as argument the channels $\overline{d}$ and c. Channel forwarding compilation is non-trivial and detailed in Section 3.3.

Our design choice requires addressing the differences in the typing discipline for channels between Go and our language. In Go, a channel's type codifies the type of values that can exchanged over the channel, with this type being fixed at channel creation. In our session-typed language, a channel's payload changes over time as the session protocol is carried out. However, in the term

`send` c 4 ; `send` c `true` ; `close` c, where c has type $int \wedge bool \wedge \mathbf{1}$, the payload of channel c is at first an integer, followed by a boolean value, and a termination message. This kind of pattern cannot be implemented directly using a simple channel type in Go. To solve this issue, we take advantage of the type information that is collected and integrated into the abstract syntax tree during the type checking stage and encode a single session channel using multiple Go channels, inspired by the translation of session types into linear types [5]. More precisely, we use a different Go channel per session *operation*, such that each operation involves not only sending the specified payload but also the *channel* on which the next action of the session protocol will take place. We thus leverage Go's static typing (obtaining an additional guarantee of end-to-end correctness), instead of relying on runtime type casts which would incur in a run time cost.

Encoding Session Channels in Go. Since session types are inherently stateful objects, where each communication action advances the session to a next state (e.g. communicating an integer over a session channel of type $int \wedge bool \wedge \mathbf{1}$ advances the session type to $bool \wedge \mathbf{1}$, where a value of type *bool* must be exchanged), we encode each session type that is used in a program into a set of protocols or session type states which are represented using custom Go types, associating to those types methods implementing the appropriate communication actions. While not all channels are explicitly type-annotated, this information is reconstructed and added to the program's abstract syntax tree during type checking.

In the case of a data exchange, the custom type contains information about the type name, the type of the exchanged data, and information about the next equivalent state in the state machine. In the case of a choice session type, instead of just information about the next type, the correspondent custom type contains a mapping of labels to their respective next states. For a terminal session type, there is no next type, since communication ends.

Associated to each type are transition methods that advance the session, as well as the initialization functions of the types. The following is a snippet of the code generated for the type $int \wedge bool \wedge \mathbf{1}$, corresponding to the initial state:

```
type _send_int struct { c chan int ; next *_send_bool }
func init_send_int() *_send_int { return &_send_int{make(chan int), nil} }
func (x *_send_int) Send(v int) *_send_bool { ... }
func (x *_send_int) Recv() (int, *_send_bool) { ... }
```

The initial protocol state, codified by the Go struct type `_send_int`, contains a channel on which to exchange an integer value and a reference to the next protocol state, whose type is suggestively named as `_send_bool`. Since the session type specifies a value communication, `Send` and `Recv` methods are associated with the type, performing the appropriate (type-safe) communication on the underlying Go channel and returning the next state accordingly. State initialization is performed *lazily*, and so each `init` function only initializes the outer state, with the next states being initialized by the `Send` and `Recv` operations as needed. This prevents unnecessary channel creation and provides a simpler initialization in the presence of recursive types.

To illustrate the challenge of representing recursive session types, recall type `IntCStream` from Section 2, defined as type `rec x.&{next:int^x, stop: 1}`.

Such a type codifies a protocol in which, first, a message label is exchanged. If the label is `stop`, the protocol terminates; if the label is `next`, an integer value is exchanged and the protocol repeats. As mentioned above, we compile a choice session using a message-label map, where each label is associated with an appropriately initialized representation of the corresponding branch type:

```
type _choice struct { c chan string ; ls map[string]interface{} }
func init_choice() *_choice {
    m := make(map[string]interface{})
    m["next"] = init_send_int(); m["stop"] = init_close();
    return &_choice{make(chan string), m} }
```

The Go type that corresponds to the `next` branch then contains an indirect reference to the choice type, allowing for the protocol to repeat: `type _send_int struct { c chan int ; next *_choice }`.

The main issue with this representation arises from the fact that in Go, types are *nominal*, whereas session types are treated *structurally*. As mentioned in Section 3.1, the session type `IntCStream` must be identical to `rec x.&{next:int^rec x.&{next:int^x, stop: 1}, stop: 1}` during type checking. This means that in our Go representation of sessions, we should map both `IntCStream` and all its recursive unfoldings to the Go type `_choice` above. To achieve this, we maintain during the compilation process a mapping of session types to Go types that is *quotiented* by session subtyping, effectively identifying all unfoldings of a given recursive type with the same Go type, and allowing us to reuse the same type declarations for all possible recursive usages of a recursive type. This way, all structurally identical types match the same Go type declaration. This mapping also enables an optimization in which we can reuse Go type declarations for identical session types used throughout a program.

3.3 Forwarding

The compilation of the channel forwarding construct `fwd` d c is directed by the (equal) session type of channels c and d. For instance, a forwarder between an ambient channel d of type $\mathit{int} \wedge \mathbf{1}$ and an offered channel c of the same type, typed as $d{:}\mathit{int} \wedge \mathbf{1} \vdash \texttt{fwd}\ d\ c :: c{:}\mathit{int} \wedge \mathbf{1}$, must necessarily receive from d an integer and send that integer along c, wait for the termination of d and then terminate the session on c. Thus, the compilation of the forwarder primitive synthesizes a process expression by analyzing the forwarded type. This process redirects messages from the ambient channel to the offered channel, and vice-versa (e.g. if the forwarded type is an input, the forwarder first receives on c and then sends along d). If the forwarded type is a recursive type, the forwarder code is simply embedded inside an infinite loop, taking some care for correct variable usage across loop iterations and with termination breaking out of the loop.

We highlight the recursive `next` branch from the forwarder used in the definition of `nats` from Section 2 generated by our compiler (up to renaming for readability), where Go variables c and d are used according to the specified type and subsequently restored with the appropriate values before the next iteration:

```
for { dc1 := c.Recv(); d.Send(dc1)
        switch dc1 { case "next":    d0 := d.ls["next"].(*_state_next)
                                     c1 := c.ls["next"].(*_state_next)
                                     c1d0, c1_d0 := d0.Recv()
                                     d = c1_d0
                                     c = c2.Send(c2d0)
                     case "stop": ... } }
```

4 Examples

We illustrate the expressiveness of our language with a (simplified) implementation of a concurrent cryptocurrency miner and a map-reduce style generalization. With cryptocurrency, double-spending of currency is prevented by a so-called *proof of work* function, which is computationally hard to generate but easy to verify, which is used to ensure consistency across the operation ledgers in the network. Clients try to generate an answer to the proof of work, validating their version of the ledger, and collecting currency as they do. The problem, while computationally hard, naturally suggests a concurrent solution since the calculation is performed over (disjoint) ranges of values.

In our language, we can emulate this kind of concurrent pattern with a *server* which performs the calculation over a *range of integers*, from 0 to a given value `x`, spawning four worker processes among which it distributes the work:

```
w1 <- spawn (worker ());                    send w1 0; send w1 (x / 4);
...                                         ...
w4 <- spawn (worker ());                    send w4 (3 * x / 4); send w4 x;
```

The worker processes, shown below, compute a solution over the given range. Function `partial_solve` encodes the calculation over the range, returning an integer result that is then sent to the server.

```
let worker : unit -> {int => int => int ^ 1} = fun u ->
w <- { low <- recv w; high <- recv w; send w (partial_solve low high); close w }  end;
```

The server subsequently receives the results from the workers and aggregates them using a `solve` function, sending the result back to the client:

```
res1 <- recv w1; ... ; res4 <- recv w4; send c (solve res1 res2 res3 res4); close c
```

A related concurrent pattern that is straightforward to implement in our language is a generalization of the example above, but where clients can send *a function* to be executed by a server.

A process which models such a server is shown below, receiving some integer function `myf`, which takes two integers and returns a single integer as a result. The server receives the function and both its arguments before executing it and returning the result. We omit a concurrent execution of the received function for the sake of space, but it is relatively easy to combine the two patterns.

```
let server : unit -> {(int->int->int) => int => int => int ^ 1} = fun u ->
c <- {myf <- recv c; fst <- recv c; snd <- recv c; send c (myf fst snd); close c} end;
```

5 Related Work

Session types in general purpose languages. Integrating session types in general purpose languages is challenging due to the need to enforce a linear typing discipline in a non-linear typing system [20]. Thus, most implementations in this setting manifest as libraries which forego most of the compile time correctness guarantees of session types, only providing runtime correctness assurances when the library API is used correctly, but with no compile time means of ensuring the usage is correct. This pragmatic approach is often complemented by a protocol description DSL, from which the API is generated. This approach has been used in the context of Scala [24], Java [13,27], F# [17], and Python [18].

When the type system of the host language is powerful enough, linearity can be *statically* encoded using type-level programming features. This has given rise to statically verified implementations in OCaml [14], Haskell [19,16,23,15], and, more recently, Rust [4]. These implementations often suffer from usability issues due to the type-level encoding of linearity giving rise to inscrutable error messages, as well as suffering from a lack of integration with the language's development ecosystem, which generally is not suited to interact with type-level programming features used in the implementations.

Natively session typed languages. Generally, languages natively featuring session types are developed in an academic setting (such as our own) and their programs are *interpreted* rather than *compiled*, since the emphasis is on the implementation of the type-checker. We highlight the language of [10], which features *refined* session types. This interpreted language is based on a classical formulation of session types [12] and so has weaker static correctness properties when compared to our own. This is also the case with the more recent FreeST language [1] which is based on a context-free extension of session types. Both languages do not enforce deadlock-freedom by typing, only absence of communication errors.

The line of work around the session-based functional languages SILL [21,26] and Rast [7,8,6] is the most closely related to ours. Both languages are interpreted. In the former, the implementation focuses on a polarized view of session types which uniformly incorporates synchronous and asynchronous communication. In the latter, the emphasis is on extensions to the core session typing discipline that enable automated amortized parallel complexity analysis. In terms of compiled languages in this space, we highlight Concurrent C0 [29], a type-safe C-like language, with session-typed communication over channels. Concurrent C0 compiles to C or Go, but features a form of asynchronous communication that requires a specialized runtime that is absent from our communication model that can be faithfully modelled using Go's channel-based primitives directly.

Future Work. We plan to perform an extensive performance evaluation of our implementation, including exploring alternatives choices in the compilation to Go (e.g. eager vs lazy channel creation and single vs multi-channel encoding) and their impact in terms of performance. In terms of theoretical work, we aim to formalize the correctness of our compilation to Go.

Acknowledgements This work was supported by FCT/MCTES grant NOVALINCS/BASE UIDB/04516/2020.

References

1. Almeida, B., Mordido, A., Vasconcelos, V.T.: Freest: Context-free session types in a functional language. In: PLACES@ETAPS. EPTCS, vol. 291, pp. 12–23 (2019)
2. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer (2010)
3. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theor. Comput. Sci. **232**(1-2), 133–163 (2000)
4. Chen, R., Balzer, S., Toninho, B.: Ferrite: A judgmental embedding of session types in rust. In: ECOOP (2022), to appear
5. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: PPDP. pp. 139–150. ACM (2012)
6. Das, A., Hoffmann, J., Pfenning, F.: Parallel complexity analysis with temporal session types. Proc. ACM Program. Lang. **2**(ICFP), 91:1–91:30 (2018)
7. Das, A., Pfenning, F.: Rast: Resource-aware session types with arithmetic refinements (system description). In: FSCD. LIPIcs, vol. 167, pp. 33:1–33:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
8. Das, A., Pfenning, F.: Session types with arithmetic refinements. In: CONCUR. LIPIcs, vol. 171, pp. 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
9. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5), 98:1–98:38 (2021)
10. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined session types. In: SEFM Workshops. Lecture Notes in Computer Science, vol. 8368, pp. 15–28. Springer (2013)
11. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: ESOP. Lecture Notes in Computer Science, vol. 1576, pp. 74–90. Springer (1999)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998)
13. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. pp. 116–133 (2017). `https://doi.org/10.1007/978-3-662-54494-5_7`, `https://doi.org/10.1007/978-3-662-54494-5_7`
14. Imai, K., Yoshida, N., Yuen, S.: Session-ocaml: A session-based library with polarities and lenses. Sci. Comput. Program. **172**, 135–159 (2019). `https://doi.org/10.1016/j.scico.2018.08.005`, `https://doi.org/10.1016/j.scico.2018.08.005`
15. Kokke, W., Dardha, O.: Deadlock-free session types in linear haskell. In: Haskell. pp. 1–13. ACM (2021)
16. Neubauer, M., Thiemann, P.: An implementation of session types. In: Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings. pp. 56–70 (2004). `https://doi.org/10.1007/978-3-540-24836-1_5`, `https://doi.org/10.1007/978-3-540-24836-1_5`
17. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138 (2018). `https://doi.org/10.1145/3178372.3179495`, `https://doi.org/10.1145/3178372.3179495`

18. Neykova, R., Yoshida, N.: Multiparty session actors. Logical Methods in Computer Science **13**(1) (2017). https://doi.org/10.23638/LMCS-13(1:17)2017, https://doi.org/10.23638/LMCS-13(1:17)2017
19. Orchard, D.A., Yoshida, N.: Effects as sessions, sessions as effects. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 568–581 (2016). https://doi.org/10.1145/2837614.2837634, https://doi.org/10.1145/2837614.2837634
20. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017). https://doi.org/10.1017/S0956796816000289, https://doi.org/10.1017/S0956796816000289
21. Pfenning, F., Griffith, D.: Polarized substructural session types. In: FoSSaCS. Lecture Notes in Computer Science, vol. 9034, pp. 3–22. Springer (2015)
22. Pierce, B.C., Turner, D.N.: Local type inference. In: POPL. pp. 252–265. ACM (1998)
23. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008. pp. 25–36 (2008). https://doi.org/10.1145/1411286.1411290, https://doi.org/10.1145/1411286.1411290
24. Scalas, A., Yoshida, N., Benussi, E.: Effpi: verified message-passing programs in dotty. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019. pp. 27–31 (2019). https://doi.org/10.1145/3337932.3338812, https://doi.org/10.1145/3337932.3338812
25. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: PPDP. pp. 161–172. ACM (2011)
26. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 350–369. Springer (2013)
27. Voinea, A.L., Dardha, O., Gay, S.J.: Typechecking java protocols with [st]mungo. In: Gotsman, A., Sokolova, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12136, pp. 208–224. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_12, https://doi.org/10.1007/978-3-030-50086-3_12
28. Wadler, P.: The marriage of effects and monads. In: ICFP. pp. 63–74. ACM (1998)
29. Willsey, M., Prabhu, R., Pfenning, F.: Design and implementation of concurrent C0. In: Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016. pp. 73–82 (2016). https://doi.org/10.4204/EPTCS.238.8, https://doi.org/10.4204/EPTCS.238.8