Grits: A Message-Passing Programming Language based on the Semi-Axiomatic Sequent Calculus

Adrian Francalanza[a], Gerard Tabone[a], Frank Pfenning[b]

[a] *University of Malta*
[b] *Carnegie Mellon University*

Abstract

This paper introduces Grits, a channel-based message-passing concurrent language based on the semi-axiomatic sequent calculus, a logical foundation underpinning intuitionistic session types. The language leverages modalities from adjoint logic to express a number of programming idioms such as broadcast communication and message cancellation. The Grits interpreter is developed using Go, and consists primarily of two components: a type-checker and an evaluator.

Keywords: behavioural types, concurrency, language implementation

1. Introduction

In channel-based programming languages such as Go [1], concurrent programs are susceptible to blocking bugs due to communication errors caused by mismatching payloads and missing messages [2]. Asynchrony, as in case of Erlang [3], can reduce deadlocks while increasing the risks of race conditions due to message reordering. Grits is a concurrent programming language that runs on the Go concurrency platform. It prevents deadlocks and race conditions statically via a type system that is based on the semi-axiomatic sequent calculus with adjoint logic [4, 5, 6], a logical framework specifically developed to reason about asynchronous computation.

This paper showcases how Grits [7] can be used for type-driven development [8] that tames the complexities of concurrent systems. Sec. 2 explains how behavioural types describe the communication protocols on channels, leveraging adjoint logic modalities to express concurrency patterns that depart from strict linearity. The types act as code component interfaces: they enable static error detection and facilitate compositional development. Sec. 3 describes the features of the Grits compilation and its architecture. Sec. 4 concludes. The full executable program of the excerpts discussed in this paper may be found in `https://github.com/gertab/Grits` (archived [9]). The readers are also encouraged to consult the companion paper [7] for details about the underlying theory of Grits.

2. Programming in Grits

2.1. Sax *Expressed as an Adjoint Logic*

Grits is based on the semi-axiomatic sequent calculus (Sax) [4, 10], a logical framework that blends features of the sequent calculus with an axiomatic presentation of intuitionistic logic, replacing non-invertible rules by corresponding axioms. It considers an extension based on adjoint logic [11, 5] to uniformly handle the *linear* (`lin`), *affine* (`aff`), *multicast* (`mul`) and *replication* (`rep`) modalities with combinations of the contraction (*i.e.,* copying) and weakening (*i.e.,* cancellation) substructural properties; these modalities can express a variety of common programming idioms such as broadcast communication and message cancellations. Modalities are ordered as $m \geq n$, whenever modality m has *more* substructural properties than n. *E.g.* `lin` is bottom since it neither supports contraction nor weakening whereas `rep` is top because it supports both; `aff` and `mul` only support weakening and contraction respectively.

Email addresses: `adrian.francalanza@um.edu.mt` (Adrian Francalanza), `gerard.tabone@um.edu.mt` (Gerard Tabone), `fp@cs.cmu.edu` (Frank Pfenning)

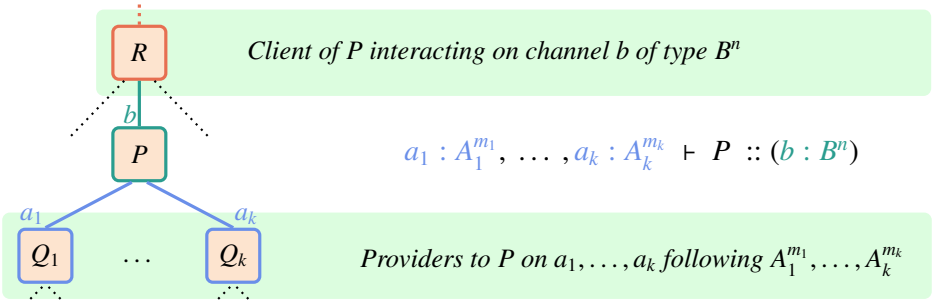

Figure 1: Hierarchical structure of processes, from P's perspective

Grits uses (session) types, A, that are indexed by a specific *modality*, m, as A^m. They can be composed of subtypes at the *same modality*, *e.g.* $A^m \multimap B^m$, with the excepting of upshifts, $\uparrow_n^m A^n$, and downshifts, $\downarrow_n^m A^m$; these delimit transitions between modalities and require that $m \geq n$. The Grits type system uses these types to statically reason about message-passing programs with a continuation-passing style of interaction. Typed processes are organised in a hierarchical structure, where a process *provides* behaviour on a channel to its parent process, and is a *client* to the channel behaviour of its children processes. The static analysis centers on the intuitionistic judgement eq. (1) below.

$$x_1 : A_1^{m_1}, \ldots, x_k : A_k^{m_k} \vdash P :: (y : B^n) \tag{1}$$

It defines an interface specification for process P, asserting that it *provides* the behaviour described by type B^n on the channel y assuming that some k processes (to which it is *client*) provide a behaviour described by type $A_i^{m_i}$ on channel x_i respectively. Eq. (1) observes the *mode independence* property: every modality in the antecedent, m_i, must allow at least as many structural properties as the modality in the succedent n, *i.e.,* $m_i \geq n$. At runtime, the variables $x_1, \ldots, x_k, y$ in eq. (1) are instantiated to dynamically-allocated channels $a_1, \ldots, a_k, b$, resulting in the process hierarchy in fig. 1, where P provides on channel b to some R and is a client to $Q_1, \ldots, Q_k$ on channels $a_1, \ldots, a_k$.

2.2. Grits *Type System and Static Programs*

The type and process syntax of Grits are shown in tbl. 1, which also includes the concrete type syntax accepted by Grits in ASCII. The process syntax is shown both from the perspective of whether a process acts as the provider, or dually, if interacting from the other end as the client; see fig. 1. The Cut typing rule is a standard cut rule extended with mode constraints. It encapsulates computation via the interaction between a spawning (client) process Q and a spawned (provider) process P, over a dynamically allocated channel x. To preserve mode independence, all modalities in the typing context Γ for P must be greater than the mode m of the type of channel x, denoted by $\Gamma \geq m$. In turn, this mode m must support the mode n of the channel provided by Q, *i.e.,* $m \geq n$. Structural rules such as Drop and Split, manifesting weakening and contraction explicitly, rely on the independence guarantees for type soundness.

$$\frac{\Gamma \geq m \quad m \geq n \quad \Gamma \vdash P :: (x{:}A^m) \quad \Gamma', x{:}A^m \vdash Q :: (u{:}B^n)}{\Gamma, \Gamma' \vdash x \leftarrow \mathsf{new}\ P; Q :: (u : B^n)}\ \textsc{Cut}$$

$$\frac{m \in \{\texttt{aff}, \texttt{rep}\} \quad \Gamma \vdash P :: (w{:}B^n)}{\Gamma, u{:}A^m \vdash \mathsf{drop}\ u; P :: (w{:}B^n)}\ \textsc{Drop} \qquad \frac{m \in \{\texttt{mul}, \texttt{rep}\} \quad \Gamma, x{:}A^m, y{:}A^m \vdash P :: (w{:}B^n)}{\Gamma, u{:}A^m \vdash \langle x, y\rangle \leftarrow \mathsf{split}\ u; P :: (w{:}B^n)}\ \textsc{Split}$$

The communication rules are defined in terms of type connectives in tbl. 1 as either left rules (when the connective is an antecedent) or right rules (when the connective is a succedent). The typing rules for output operations are defined as *axioms*, modeling asynchronous interaction. Concretely, a send operation is typed by the right axiom $\otimes$R, when it provides on a channel w described by the tensor type $A^m \otimes B^m$, *i.e.,* send a value of type A^m and continue as B^m, or by the left axiom $\multimap$L when interacting on a client channel w described the implication type $A^m \multimap B^m$, *i.e.,* once a value

Abstract Types	Concrete Types	Process Syntax (Provider)	Process Syntax (Client)
$A \otimes B$	`A * B`	send $u\langle v, w\rangle$	$\langle x, y\rangle \leftarrow$ recv $u; P$
$A \multimap B$	`A -* B`	$\langle x, y\rangle \leftarrow$ recv $u; P$	send $u\langle v, w\rangle$
$\oplus\{l : A_l\}_{l\in L}$	`+{l1 : A1, ...}`	$u.l\langle v\rangle$	case u $(l\langle y\rangle \Rightarrow P_l)_{l\in L}$
$\&\{l : A_l\}_{l\in L}$	`&{l1 : A1, ...}`	case u $(l\langle y\rangle \Rightarrow P_l)_{l\in L}$	$u.l\langle v\rangle$
$\mathbf{1}$	`1`	close u	wait $u; P$
$\uparrow^n_m A^m$	`m /\ n A`	$x \leftarrow$ shift $u; P$	cast $u\langle v\rangle$
$\downarrow^m_n A^m$	`m \/ n A`	cast $u\langle v\rangle$	$x \leftarrow$ shift $u; P$

Table 1: Abstract and concrete mapping for types and process syntax

of type A^m is received, continue as B^m. Dually, a receive operation is typed by the $\otimes$L and $\multimap$R rules.

$$\frac{}{u : A^m,\ v : B^m \vdash \mathsf{send}\ w\langle u, v\rangle :: (w : A^m \otimes B^m)}\ \otimes\mathrm{R} \qquad \frac{\Gamma, x : A^m, y : B^m \vdash P :: (w : C^n)}{\Gamma, u : A^m \otimes B^m \vdash \langle x, y\rangle \leftarrow \mathsf{recv}\ u; P :: (w : C^n)}\ \otimes\mathrm{L}$$

$$\frac{\Gamma, x{:}A^m \vdash P :: (y{:}B^m)}{\Gamma \vdash \langle x, y\rangle \leftarrow \mathsf{recv}\ w; P :: (w{:}A^m \multimap B^m)}\ \multimap\mathrm{R} \qquad \frac{}{u{:}A^m, w{:}A^m \multimap B^m \vdash \mathsf{send}\ w\langle u, v\rangle :: (v{:}B^m)}\ \multimap\mathrm{L}$$

For further details on the type system refer to [7].

A Grits program starts with a a collection of *contractive* [12] equi-recursive type definitions, $t^m = A^m$, that may be mutually dependent. This is followed by a sequence of *named* process template declarations, each subject to the type judgement of eq. (1); the static syntax refers to the channel on which a process provides via the `self` keyword. The last line of a Grits program declares the main process that starts the computation by calling a process definition.

```
type t = m A   // t^m = A^m
...
let p(x1 : m1 A1, ..., xk : mk Ak) : m B = P   // p(x_1, ..., x_k) = P  with
...                                                x_1:A_1^{m_1}, ..., x_k:A_k^{m_k} ⊢ P :: (self:B^m)
exec q()
```

2.3. *Type-Driven Development by Example in* Grits

Consider the scenario where a hospital admits insured patients via their social security number (SSN). This value is typically used by more than one entity, *e.g.* by the hospital itself to retrieve a patient's records and medical history, but also by the insurance agency to check whether the patient is covered. At the same time, an SSN contains sensitive information and should only be shared with entities that genuinely need it (and actually use it) to maintain confidentiality. A type-driven design allows us to divulge an SSN to multiple users and limit unnecessary communication of this sensitive data by detecting unnecessary SSN disclosure. An SSN is therefore assigned the `mul` (multicast) modality that allows copying for multiple uses but prohibits the discarding of disseminated SSNs via cancellations.

```
1  type ssn = mul +{ cons : (rep \/ mul digit) * ssn,  nil : 1 }
2  type digit = rep +{ _0 : 1, _1 : 1, _2 : 1, _3 : 1, ..., _9 : 1 }
```

Concretely, we define the type `ssn` as a `mul` list of digits with a *big-endian* interpretation, where the most significant number is read first, line 1. In Grits, lists are encoded as processes that provide the choice of either `cons` followed by the communication of a `digit` and another `ssn`, or a `nil` which then closes the interaction, *i.e.,* type `1` (see tbl. 1). Since the use of an `ssn` does not necessarily need to access all of its digits, a `digit` is assigned a `rep` modality to allow weakening. To delineate the change in modality, a downshift from `rep` to `mul` is used before type `digit` on line 1, *i.e.,* `(rep \/ mul digit)`. For illustrative purposes,[1] a digit type is also encoded as a process that selects a label `_n` (where `n` $\in 0..9$) and closes the channel it provides on.

[1]The current version of Grits does not support basic datatypes such as integers.

```
let hospital_admission(n : ssn) : lin +{valid : 1, not_valid : 1} // hospital interface
...
let insurance(n : ssn) : lin 1 // insurance interface
```

Our type-driven methodology facilitates a compositional approach to software development. Once the type `ssn` is defined, line 1, we can describe the interfaces for the `insurance` and `hospital_admission` processes (above) that handle the processing of an SSN. This permits the independent development of the code for the `main` process.

```
3  let main() : lin 1 =
4      ss : ssn <- new ssExample(); // contains social security number 43210
5      <ss1, ss2> <- split ss;
6
7      i <- new insurance(ss1); wait i; // SS used by insurance agency
8
9      h <- new hospital_admission(ss2); // SS used by hospital admission
10     case h (
11         valid<h'>     => print found_patient; wait h'; close self
12       | not_valid<h'> => print not_found; wait h'; close self
13     )
14
15 exec main() // Execute main
```

The `main` process that drives our program creates a new SSN on line 4 that encodes some sample number, *e.g.* 43210, and is bound to the variable `ss`. On line 5 it is copied to `ss1` and `ss2`, as permitted by modality `mul` of type `ssn` (see line 1). The first copy, `ss1`, is passed on to the `insurance` process for record keeping whereas the second copy, `ss2`, is passed to the `hospital_admission` process to determine if it is a valid SSN or not. The `main` process prints the outcome of this validation check and closes the channel it provides on. Without exposing its internals, the interface of the `hospital_admission` process informs the `main` process that it must (linearly) handle the value sent by it, modality `lin`, and be prepared to branch for a selection on either label `valid` or `not_valid`, lines 10 to 13.

```
16 let insurance(n : ssn) : lin 1 =
17     n' <- new encrypt(n);
18     ...        // code for recording encrypted SSN
19     close self
20
21 let encrypt(n : ssn) : ssn =
22   case n (
23     cons<c> =>
24       <curr_dgt, rem> <- recv c;
25       curr_rep <- shift curr_dgt; // from mul to rep
26       case curr_rep (
27           _0<c> =>
28               inv_dgt : digit <- new self._9<c>; // inverts 0 to 9
29               inv_mul : rep \/ mul digit <- new cast self<inv_dgt> // from rep to mul
30               rem_inv : ssn <- new encrypt(rem);
31               inv : mul ((rep \/ mul) digit) * ssn <- new send self<inv_mul,rem_inv>;
32               self.cons<inv>
33         | _1<c> => ...  // inverts 1 to 8
34         | ...
35       )
36    | nil<c> => self.nil<c>
37   )
```

The `insurance` process, line 16, encrypts an SSN before recording it. Our encryption, line 21 traverses the SSN and inverts *every* digit, see line 28 for case `_0<c>`. *E.g.* 43210 is encoded as 56789. Note how a digit needs to be (down)shifted before it operated on, line 25, and then cast (*i.e.,* upshifted), line 29 to be composed back as an SSN.

Grits's behavioural type-driven approach guides the implementation: from type `ssn` on line 21, the `encrypt` process knows it needs to input a digit after branching on a `cons` selection, line 24, and that this digit needs to be shifted from `mul` to `rep` before it is used, line 25. Omitting any of these commands immediately results in a type-checking error.

```
38 let hospital_admission(n : ssn) : lin +{valid : 1, not_valid : 1} =
39   res : mul +{even : 1, odd : 1} <- new even_odd(n); // check if n is even
40   case res (
41       even<c> =>
42           ...   // wait on c and spawn a process providing on c'
43           self.valid<c'>
44     | odd<c> =>
45           ...   // wait on c and spawn a process providing on c'
46           self.not_valid<c'>
47   )
48
49 let even_odd(n : ssn) : mul +{even : 1, odd : 1} = // if n is even or odd
50   c : rep 1 <- new close self;
51   d : digit <- new self._0<c>; // dummy digit
52   even_odd_inner(n, d)
53
54 let even_odd_inner(n : ssn, prev : digit) : mul +{even : 1, odd : 1} =
55   case n (
56       cons<n'> =>
57           <curr, tail> <- recv n';
58           curr_rep <- shift curr;
59           drop prev;      // may be discarded in rep mode
60           even_odd_inner(tail, curr_rep)
61     | nil<n'> =>
62           wait n';
63           even_odd_digit(prev) // check the last digit
64   )
65
66 let even_odd_digit(d : digit) : mul +{even : 1, odd : 1} =
67   case d (
68       _0<c>  => wait c;
69                 c' : mul 1 <- new close self;
70                 self.even<c'>  // select even branch
71     | _1<c>  => ...
72                 self.odd<c'>   // select odd branch
73     | _2<c>  => ...
74     ...
75 )
```

For our example, the `hospital_admission` process deems an SSN valid when it is even, lines 39 and 41. This check involves traversing an SSN to its last digit, line 49, to check if it is even using process `even_odd_digit`, line 66. The traversal is carried out by the process `even_odd_inner`, line 54; it assumes that an SSN is at least one digit long, which explains the dummy digit initialisation on line 51. Importantly, since digits are typed at modality `rep`, `even_odd_inner` may cancel (*i.e.,* using `drop`) the digits, line 59, preceding the last one (which is used), line 63.

```
76 // Initialize each process individually
77 prc[ssn1, ssn2] : ssn = ssExample() // Implicit splitting
78 prc[i] : lin 1 = insurance(ssn1)  // records kept by insurance
79 prc[h] : lin +{found : 1, not_found : 1} = hospital_admission(ssn2) // admissions check
80 prc[m] : lin 1 = wait i;
81                  case h ( ... )  // handle result obtain from hospital admission
```

To facilitate debugging, Grits permits individual process instantiation with channel binding to recreate a specific concurrency interleaving, instead of having to rely on the `main` launcher process to regenerate the interleaving from scratch. The snapshot on lines 77–81 initiates execution with multiple processes that provide on the channels `ssn1`, `ssn2`, `i`, `h` and `m`. Lines 77 creates two processes running `ssExample()`, providing on `ssn1` and `ssn2` respectively.

```
82  assuming ssn1 : ssn, // process providing on ss1 is assumed to have type ssn
83           h : lin +{found : 1, not_found : 1}
84  prc[i] : lin 1 = insurance(ssn1)
85  prc[m] : lin 1 = wait i;
86                   case h ... // handle result obtain from the hospital admission process
```

A type-driven approach also permits incremental (typed) development in Grits, typechecking snippets without having to write the full codebase. Using the keyword `assuming`, the code above (lines 82–86) does not define the precise computation code for the processes providing on the channels `ssn1` and `h`. Yet, by describing their type (interface), we can still type-check the incomplete program, *e.g.* `ssn1` is assumed to have type `ssn` on line 82.

3. Using Grits

Grits is a CLI application offering several flag-based features to configure its interpretation pipeline. To type-check and execute a program `file.grits` one can use the following command, where `grits` is the executable version of our tool:

```
$ ./grits path/to/file.grits
$
$ Typecheck successful
$ Spawning 2 processes...
```

Grits performs static analysis based on the type system described in sec. 2.2. It reports type errors with informative messages, including the line number where each issue occurs. For instance, the type errors discussed in sec. 2.3, such as invalid mode shifting or improper name usage (*e.g.* omitting a required receive action), are automatically detected.

Consider the main process from sec. 2.3, which correctly handles *linear* results returned from insurance and hospital services. Suppose we erroneously modify this process by ignoring the hospital result received over channel `h`, replacing lines 10–13 with

```
10  drop h; close self // erroneously dropping a linear channel
```

When executed, this triggers the type-checking error shown below:

```
$ ./grits examples/social_security.grits
$ Initiating typechecking
$ (Line 10) Typechecking error in function main(); unable to drop h, which is
    ↪ in linear mode
```

The tool offers further functionality to fine-tune the computation process. For instance, the output verbosity can be adjusted using `--verbosity <level>`, where more details relating to type-checking and execution can be obtained. *E.g.* the following command type-checks a program, detailing the typing rules used.

```
$ ./grits --verbosity=3 examples/social_security.grits
```

The tool itself is compiled using the Go language as a concurrent executable, leveraging the native concurrency features provided by the Go platform in order to exploit any parallelism proffered by the underlying hardware. Go offers cross-platform compilation, making it simple to run Grits on most operating systems. We also offer a Dockerized version for a simpler testing environment, without having to install any dependencies. The Grits tool acts as an *interpreter* that takes `grits` programs as input and processes them in three parts using: a parser, a type-checker and an evaluator (akin to a tree-walk interpreter [13]); see fig. 2.

The parser uses a Yacc-based library called *goyacc*, which parses our Grits-based language into analysable Go structures [1], acting as an abstract syntax tree (AST) of the `grits` programs. The type-checker uses syntax-directed

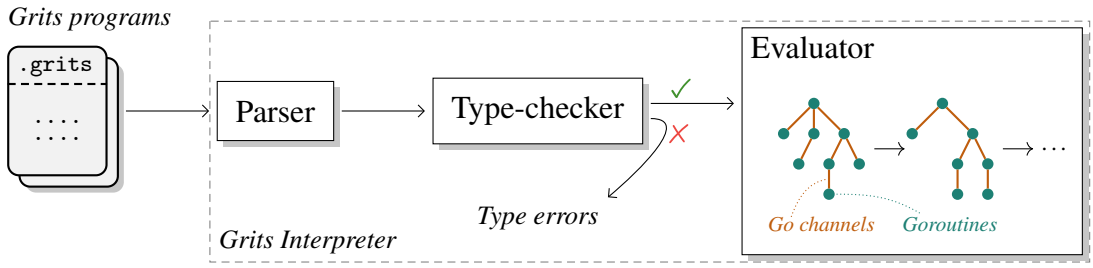

Figure 2: Pipeline for type-checking and executing programs using Grits

rules which ensure that the programs follow the specifications dictated by Sax; see sec. 2.2. Apart from reading the program to check for its well-typedness, the type-checker augments the intermediary code by annotating each name with a polarity inferred from the types. These annotations guide execution when handling forwarding via buffered channels: depending on polarity, a forwarding process either sends a request to a provider or awaits a message to forward to a client; see [7] has more details. For incomplete programs (*i.e.,* using the *assuming* keyword), one can use the `--noexecute` flag to only parse and type-check programs.

The tool leverages the concurrency abstractions offered by the Go language to execute the concurrent processes described in `grits` programs on multicore systems. The interpretation pipeline depicted in Fig. 2 translates each `grits` process to a corresponding Go coroutine (called goroutine) providing the behaviour described by the process on a dedicated Go channel. This maintains a one-to-one mapping between the concurrency units of the respective languages, *i.e.,* Grits and Go. Grits also supports two execution interpretations via the synchronous (`--sync`) and asynchronous (`--async`) communication modes, which correspond to Go's unbuffered or buffered channels, respectively; asynchrony at the level of `grits` processes (where channels are used once) ensures that the two interpretations are operationally equivalent. The two interpretations were used to conduct empirical evaluations and investigate their respective runtime performance. Benchmarking information, such as the execution timings for each execution mode were obtained using the `--benchmark` flag; see [7] for details.

4. Conclusion

We present a tool, called Grits, that interprets the semi-axiomatic sequent calculus with adjoint logic as channel-based message-passing concurrent programs in Go. The Curry-Howard correspondence links intuitionistic session types to channel-passing concurrent processes, resulting in a language that inherits desirable properties such as deadlock freedom and session fidelity. In the future we plan on extending Grits to support shorthand notation that makes the code less verbose, such as eliding `drop` commands, or writing

```
4       <ss1, ss2> : ssn <- new ssExample();
```

in lieu of lines 4–5. We also plan to extend the language to handle the notion of shared processes [14, 15, 16] that co-exists alongside replicated processes with a copy semantics.

4.1. Related Work

The closest tools to Grits are a pedagogic tool [10], and SNAX [17], which are both based on the semi-axiomatic sequent calculus. Although the pedagogic tool targets a channel-based language, it can only support linear computation (*i.e.,* no duplication or cancellation). SNAX supports all four modalities like Grits, but targets a functional programming language instead. Both tools are based on Standard ML implementations. Other tools use intuitionistic session types. Rast [18] integrates session types with arithmetic refinements, while Nomos [15] extends this for smart contracts. Nomos, along with Ferrite [16] offer a notion of type shifting, between linear and shared processes, but does not support a copy semantics. Several other tools [19, 20, 21, 22, 23] employ the *classical* binary or multiparty session types to verify message flows in existing languages. For instance, Lange et. al. [22] infers behavioral types from Go programs to ensure deadlock-free communication, addressing the limitations of Go's compiler in detecting message-passing errors [2].

Metadata

Nr.		
C1	Current code version	v1.0
C2	Permanent link to code/repository used for this code version	https://github.com/gertab/Grits
C3	Permanent link to Reproducible Capsule	https://doi.org/10.5281/zenodo.10732024
C4	Legal Code License	GNU GPLv3
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Go
C7	Compilation requirements, operating environments and dependencies	Go 1.21 or later, multi-platform (Linux, macOS, Windows)
C8	If available, link to developer documentation/-manual	
C9	Support email for questions	*gerard.tabone@um.edu.mt*

Table 2: Code metadata

Acknowledgements

This work has been supported by the Security Behavioural APIs project (No: I22LU01-01) funded by the UM Research Excellence Funds 2021, the Tertiary Education Scholarships Scheme (Malta) and IPCEI-UM (No: E24LO17-01) project under Malta Enterprise.

References

[1] Effective Go - The Go Programming Language (n.d.). URL https://go.dev/doc/effective_go#sharing

[2] T. Tu, X. Liu, L. Song, Y. Zhang, Understanding real-world concurrency bugs in go, in: I. Bahar, M. Herlihy, E. Witchel, A. R. Lebeck (Eds.), Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019, ACM, 2019, pp. 865–878. doi:10.1145/3297858.3304069.

[3] F. Cesarini, S. Thompson, Erlang Programming: A Concurrent Approach to Software Development, O'Reilly Media, 2009.

[4] H. DeYoung, F. Pfenning, K. Pruiksma, Semi-axiomatic sequent calculus, in: Z. M. Ariola (Ed.), 5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference), Vol. 167 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 29:1–29:22. doi:10.4230/LIPICS.FSCD.2020.29.

[5] K. Pruiksma, F. Pfenning, A message-passing interpretation of adjoint logic, J. Log. Algebraic Methods Program. 120 (2021) 100637. doi:10.1016/J.JLAMP.2020.100637.

[6] K. Pruiksma, Adjoint logic with applications, Ph.D. thesis, Carnegie Mellon University (6 2024). doi:10.1184/R1/25900861.v1.

[7] A. Francalanza, G. Tabone, F. Pfenning, Implementing a Message-Passing Interpretation of the Semi-Axiomatic Sequent Calculus (SAX), in: I. Castellani, F. Tiezzi (Eds.), Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings, Vol. 14676 of Lecture Notes in Computer Science, Springer, 2024, pp. 295–313. doi:10.1007/978-3-031-62697-5_16.

[8] E. Brady, Type-Driven Development with Idris, Manning Publications, 2017.

[9] G. Tabone, Grits: Implementing a Message-Passing Interpretation of the Semi-Axiomatic Sequent Calculus (Sax) (artefact for Coordination'24), https://github.com/gertab/Grits (May 2025). doi:10.5281/zenodo.10837897.

[10] F. Pfenning, Lecture notes on Semi-Axiomatic Sequent Calculus, Course notes for Substructural Logics (15-836). Accompanying tool available from https://www.cs.cmu.edu/~fp/courses/15836-f23/resources.html (2023).

[11] K. Pruiksma, W. Chargin, F. Pfenning, J. Reed, Adjoint logic, Unpublished manuscript, April (2018).

[12] S. J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Informatica 42 (2-3) (2005) 191–225. doi:10.1007/S00236-005-0177-Z.

[13] T. Ball, Writing an Interpreter in Go, Thorsten Ball, 2018.

[14] S. Balzer, F. Pfenning, Manifest sharing with session types, Proc. ACM Program. Lang. 1 (ICFP) (2017) 37:1–37:29. doi:10.1145/3110281.

[15] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, I. Santurkar, Resource-aware session types for digital contracts, in: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021, IEEE, 2021, pp. 1–16. doi:10.1109/CSF51468.2021.00004.

[16] R. Chen, S. Balzer, B. Toninho, Ferrite: A judgmental embedding of session types in rust, in: K. Ali, J. Vitek (Eds.), 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany, Vol. 222 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 22:1–22:28. doi:10.4230/LIPICS.ECOOP.2022.22.

[17] Lecture notes on Adjoint Functional Programming for Pfenning's course at OPLSS 2024, accompanying tool available from https://bitbucket.org/fpfenning/snax (2024). URL https://www.cs.uoregon.edu/research/summerschool/summer24/lectures/pfenning3.pdf

[18] A. Das, F. Pfenning, Rast: A language for resource-aware session types, Log. Methods Comput. Sci. 18 (1) (2022). doi:10.46298/LMCS-18(1:9)2022.

[19] S. Fowler, An Erlang implementation of multiparty session actors, in: M. Bartoletti, L. Henrio, S. Knight, H. T. Vieira (Eds.), Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016, Vol. 223 of EPTCS, 2016, pp. 36–50. doi:10.4204/EPTCS.223.3.

[20] R. Neykova, N. Yoshida, Multiparty session actors, Log. Methods Comput. Sci. 13 (1) (2017). doi:10.23638/LMCS-13(1:17)2017.

[21] A. Scalas, N. Yoshida, Lightweight session programming in scala, in: S. Krishnamurthi, B. S. Lerner (Eds.), 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy, Vol. 56 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 21:1–21:28. doi:10.4230/LIPIcs.ECOOP.2016.21.

[22] J. Lange, N. Ng, B. Toninho, N. Yoshida, A static verification framework for message passing in go using behavioural types, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1137–1148. doi:10.1145/3180155.3180157.

[23] A. Francalanza, G. Tabone, ElixirST: A session-based type system for Elixir modules, J. Log. Algebraic Methods Program. 135 (2023) 100891. doi:10.1016/J.JLAMP.2023.100891.