Bachelor's Thesis Computing Science

# Achieving multiparty deadlock-free communication through linear typing

*A Haskell implementation of the MPGV language*

Menzo van Kessel
s1050954

July 10, 2023

*First supervisor/assessor:*
Dr. Robbert Krebbers

*Assistant supervisor:*
Jules Jacobs MSc.

*Second assessor:*
Dr. Freek Wiedijk

Radboud University

## Abstract

Session types allow us to encode channel-based communication protocols on the type level. This way, we can perform static analysis on communication, e.g. to ensure deadlock-freedom. Since protocols are stateful, we need to add state to the type system. For this, linear types can be used.

MPGV is a (theoretical) multiparty communication language using linear types to ensure protocol adherence. However, no implementation has been made before. One of the main challenges for writing an implementation is making the MPGV typing rules more computable, and to remove the need for unnecessary type annotations. This thesis presents MPGVR, an implementation of the MPGV language. MPGVR uses an algorithmic translation of the typing rules and uses bidirectional type checking to solve both problems.

Other contributions described in this thesis include several practical extensions to MPGV, and a formalisation for shadowing variables in typing judgements.

## Acknowledgements

I wish to thank my supervisors for their ample and regular feedback. From the content-related feedback to remarks about inconsistent writing, all was appreciated. I would also like to thank my family for their love and moral support, through the good times and the bad times.

## Artifact

The MPGVR implementation can be found at [22].

# Contents

Chapter 1.

# Introduction

Concurrency is one of the many tools in a programmer's toolbox. From modelling asynchronous tasks to increasing performance, concurrency shows up everywhere. With the amount of cores in CPUs steadily increasing [19] and CPU frequency stagnating over the last years, we see that utilising multiple threads is becoming more and more important.

However, managing shared resources and thread communication without introducing problems like race conditions and deadlocks remains a tough problem to solve. High-level constructions such as (mutex) locks and message passing simplify communication to make the process less error-prone.

In this thesis, we focus on (channeled) message passing systems. Many languages have libraries for channel-based communication. For example, take a look at Rust's MPSC channels[1]:

```
 1 use std::sync::mpsc::channel;
 2 use std::thread;
 3
 4 let (tx, rx) = channel<u32>();
 5
 6 thread::spawn(move|| {
 7     tx.send(7106412).unwrap();
 8 });
 9
10 assert_eq!(rx.recv().unwrap(), 7106412);
```

In this example, we first create a channel (line 4) with two endpoints. Each thread will use one of these endpoints. We then create a thread (line 6) that uses `tx` to send a number (line 7). The main thread then uses `rx` to receive this number (line 10), checking that it is correct.

With channels, there are a few points at which the construction can fail:

(a) Messages are sent with a type the receiver does not expect;

(b) Messages are sent in the wrong order;

(c) The channel is still being used when its resources have already been released;

(d) Messages are not being sent, leading to a deadlock on the receiving side.

To show how languages could solve these problems, we consider the Rust language again. It solves problem (a) by making the channel typed; only one type of value can be sent over the channel, meaning that the receiver knows what to expect. It also solves problem (c) because Rust has strict ownership rules; once a channel is freed, the compiler can verify whether it will not be used again. Problem (b) may occur, if you are sending two values of the same type. When using different types, however, you'll have to use two separate channels (because of the hardcoded type on a channel). In that case, the receiver can simply control the order of receiving messages by deciding when to consume from each channel. However, problem (d) still exists. Rust allows a variable to not be used, meaning we can write something like this:

[1]Example inspired by the documentation: https://doc.rust-lang.org/std/sync/mpsc/index.html

```rust
use std::sync::mpsc::channel;
use std::thread;

let (tx, rx) = channel<u32>()

thread::spawn(move|| {
    // we have forgot to send something
});

assert_eq!(rx.recv().unwrap(), 7106412);
```

Now, the main thread will be deadlocked.

We see that types and ownership rules are promising constructs to fix certain channel problems with. But, can we solve more problems than just (a) and (c)? To find the answer, we will look into the field of *session types*. This is a field where channeled communication is encoded into types, such that we can derive certain guarantees from them.

The groundwork for session types was laid by Honda et al. [7, 8], where a process-calculus was introduced for channel-based communication. In this calculus, two parties could communicate over a bidirectional channel, and communication was encoded using *dual* types. With dual types, one side $A$ can define how they will communicate with $B$, and from that $B$ can derive how they should communicate with $A$.

For example, suppose $A$ wishes to ask $B$ whether a number is even. $A$ could use a channel with type `!N. ?B. End`. This indicates that $A$ wishes to send their natural number to $B$, then receive a boolean result from $B$, and then stop communication. This means that side $B$ must receive this natural number, and return a boolean. Therefore, we can swap each send and receive operation to get the dual type `?N. !B. End`. From this we see that problems (a) and (b) are solved by these session types, like in Rust — for (b) again assuming messages use enough different types.

From this work by Honda et al., two branches have spawned. Gay and Vasconcelos lifted session types into the world of λ-calculi [5, 6], allowing use of session types in functional programs. They introduced the *linearity* constraint on channels. This means that channels must be used exactly once. By returning new channels on every send and receive action, we guarantee that we never forget to send a message, since this newly returned channel must be used. As an example, our is-even function could be written like this:

```haskell
isEven :: ?N. !B. End -> ()
isEven c = let (n, c) = receive c in       -- c gets shadowed to type !B. End
           let c = send (n % 2 == 0) c in  -- c gets shadowed to type End
           close c                         -- returns ()
```

The calculus by Gay and Vasconcelos supports sending and receiving of any arbitrary type. This includes lists like `List[N]`, other channels like `!N. End` or any combination like `List[!List[N]. End]`. In contrast, the session types by Honda et al. needed separate constructions for sending basic values and sending channels.

Walder later formalised the calculus by Gay and Vasconcelos, turning it into the “GV” language [23]. One of the main changes is to the construction of new threads. This was changed because the original thread creation construction by Gay and Vasconcelos was not deadlock-free. Wadler proved that the GV language, using this new construction, is actually deadlock-free — which is a significant result, since it means that session types can also solve problem (d).

Another branch of session types is that of *multiparty* session types (MPST), where Honda et al. extended the two-party process-calculus to support an arbitrary amount of parties [9, 10]. This is done by generalising the dual types into a system of *global* and *local* types. Now, communication as a whole is encoded into a global type; it states which participants send/receive what to/from whom. from this global type, local types can be generated that can tell a participant what to do. For example, suppose we have a participant 0, who lets participant 1 add to a number and have participant 2 check if the result is even, after which participant 2 sends that result back to participant 0. This could be encoded as

`[0->1]N. [1->2]N. [2->0]B. End.`

The local types can then be generated:

```
A :    ![1]Nat. ?[2]B. End
B :    ?[0]Nat. ![2]N. End
C :    ?[1]Nat. ![0]B. End
```

These global types help ensure deadlock-freedom in this language. However, because of the way threads are spawned in MPST, deadlock-freedom is only guaranteed if there is only one session/global type active at once. This means that we cannot allow dynamic thread creation. This is in contrast to GV, where we can create as many two-party sessions as we like.

Recently, the MPGV language was introduced [12]. As the name might suggest, this language combines the two session type branches to introduce a multiparty deadlock-free $\lambda$-calculus. It accomplishes this by extending the GV language with the global and local type constructs from Honda et al. MPGV notably allows for dynamic thread creation, like GV, effectively removing the deficit present in MPST. MPGV is a significant discovery, since it means that the different theory on session types is compatible with eachother. As an example, we show how we could write MPGV code for the multiparty process described earlier[2]:

```
gtype GT = [0->1]N. [1->2]N. [2->0]B. End

let funB :: GT|1 -> ()
  = \c.                -- c has type ?[1]N. ![2]N. End
    n = c.receive(); -- c gets shadowed to type ![2]N. End
    c.send(n + 3);   -- c gets shadowed to type End
    c.close()        -- returns ()

let funC :: GT|2 -> ()
  = \c.                    -- c has type ?[2]N. ![0]B. End
    n = c.receive();     -- c gets shadowed to type ![0]B. End
    c.send(n % 2 == 0); -- c gets shadowed to type End
    c.close()            -- returns ()

print
  -- c has type ![1]N. ?[2]B. End
  let c = fork<GT>(c. funB c, c. funC c) in
  c.send(7106412);       -- c gets shadowed to type ?[2]B. End
  isEven = c.receive(); -- c gets shadowed to type End
  c.close();             -- closes c, discards ()
  isEven                 -- the isEven result is printed
```

There has been effort in the programming sphere to develop software implementing these various session type constructions. Many libraries are based on GV [4, 13, 14], and there exist implementations of MPST [15]. (For more discussion on these implementations, see chapter 6.) However, before this thesis, there did not exist an implementation of MPGV yet.

This thesis introduces MPGVR (short for MPGV REPL), an implementation for MPGV. For this implementation, we have three goals:

- It must be fully compatible with the original operational semantics of MPGV by Jacobs et al.;
- The type checker must implement all safety guarantees of MPGV;
- The mathematical syntax must be translated into an ASCII syntax that a typical programmer may expect.

In the process of meeting these goals, the following contributions have been made:

[2]A proper introduction to the MPGV syntax can be found in chapter 2.

- A parser, type checker and interpreter for MPGV has been written. All features of the language have been implemented, apart from recursive typing. This is left as future work. Notable features of MPGV that have been implemented are linear typing, global and local types, and type projection.
- The bidirectional typing strategy from Dunfield and Krishnaswami [3] has been used in the MPGV type checker, using the algorithmic type checking principles from Advanced Topics in Types and Programming Languages [18]. This strategy has been shown to work on a range of examples. Bidirectional typing eliminates many of the type annotations otherwise needed to typecheck an MPGV program.
- Syntactic changes to the MPGV language have been made to make it easier to write programs with. Notable changes are adding more concise channel operations akin to Haskell's do-notation, and adding concise syntax for local types where only one type of value is always ("choicelessly") sent.
- MPGV has been extended to work as a REPL. This feature gives rise to embeddability, which distinguishes this implementation from most other session type systems, which are often designed as libraries.
- A formalisation is made that allows adding shadowing to typing judgements and operational semantics with no extra effort compared to assuming Barendregt's Variable Convention [2, 20, 21]. This formalisation is used in the implementation.

This thesis is divided into the following chapters:

- In chapter 2, we go over the syntax of the MPGV language by way of showing code snippets. These snippets also show the features and usability of the language.
- In chapter 3, we go over neccessary background information one needs, in order to understand the inner workings of the interpreter described later. This information includes formal type checking of unrestricted lambda calculi (3.1), linear lambda calculi (3.2), and a mix of both calculi (3.3). We also formalise the concept of shadowing (3.4) and introduce bidirectional type checking for algorithmic typing (3.5).
- In chapter 4, we talk about the formalisations of MPGV, first formally introducing the syntax (4.1). Then, we go over the design choices and difficulties with translating regular typing rules into algorithmic ones (4.2). Lastly, we do the same, but this time translating the algorithmic rules into bidirectional ones (4.3).
- In chapter 5, we talk about the implementation details of MPGVR. First, we describe the monads used (5.1). Then, we explain design choices and difficulties writing a type checker for MPGV (5.2), based on the bidirectional typing rules. After that, we do the same talking about designing the interpreter (5.3), based on the operational semantics. We also briefly talk about the parser (5.4). Lastly, we explain the design choices and difficulties designing a REPL for MPGV (5.5).
- Chapters 6 and 7 are for related work and the conclusion, respectively.
- Appendix A contains an exhaustive list of all different typing rules and operational semantics related to MPGV.
- Appendix B contains a formal grammar of MPGVR, which is what the parser implements.

Chapter 2.

# MPGV by example

To introduce the MPGV language, and the syntax of MPGVR, this chapter provides a range of code examples. Some examples come from related work, and show that a translation from these examples into working MPGVR code is possible. The "round trip add" examples in section 2.2 are from Multiparty GV [12], and the "two buyers" example in section 2.4 is from Multiparty Asynchronous Session Types [9].

## 2.1. Factorial

Since MPGV is a functional language, let us start with a classic example: the factorial function.

```
let fac :: N -> N =
  rec fac n.
    if n > 0 then
      n * fac (n - 1)
    else
      1
```

As you can see, we have defined `fac` as a recursive function, using the `rec f x. e` construction. A recursive function is the same as a regular lambda function `\x. e`, except for the `f` variable that allows the function to call itself. The type of this function is `N -> N`, which indicates a function that transforms a natural number into another natural number. Note that subtraction rounds negative results back to zero, since we use natural numbers.

## 2.2. Round trip add

We use this section to introduce MPGV's deadlock-free concurrency system. To start, we must define a communication protocol using so-called *global types*. If communication can be encoded into a global type, it will be deadlock-free.

A global type is a series of steps, that state what value to send from which thread to which other thread. Concretely, they follow the pattern `[p->q]T.GT`, where `p` and `q` are participants, `T` is the type of data to send, and `GT` is another global type, which holds the next step(s) in the communication. Communication ends at the global type `End`.

For some basic examples using concurrency, we look at those written in the Multiparty GV paper [12]. Consider a program that sends a number to various threads. Each thread adds to the received number, then sends the result to another thread. We start at the main thread, and end at the main thread, where we print the result.

To encode this communication between three threads, we make the following global type:

```
-- Send a number around in a circle
gtype GT = [0->1]N. [1->2]N. [2->0]N. End
```

Thread 1 can then be modeled as a function that takes a communication endpoint – a *channel* – and performs its side of the communication according to the global type:

```
-- Thread that adds 3 to the number
let f1 :: GT|1 -> () =
  \c.
    n = c.receive[0](); -- ?[0]N. ![2]N. End
    c.send[2](n+3);     -- ![2]N. End
    c.close()           -- End
```

The types used here are called *local types*. They are types that encode communication from the perspective of one participant. This is in contrast to global types, which oversee communication of all participants.

A local type either receives from (`?[p]T.LT`), or sends to (`![p]T.LT`) participant `p` a value of type `T`, to then continue with local type `LT`. Like with global types, communication ends at `End`. To generate a local perspective from a global type, the *projection* operation `GT|p` is provided. In the case of this example, we would get

$$\begin{aligned}\texttt{GT|0} &= \texttt{![1]N. ?[2]N. End}\ ,\\ \texttt{GT|1} &= \texttt{?[0]N. ![2]N. End}\ ,\\ \texttt{GT|2} &= \texttt{?[1]N. ![0]N. End}\ .\end{aligned}$$

Every time a local operation is called, like `receive`, the used channel "changes" type, as indicated by the comments. In reality, since variables are immutable, the variable does not change type, but a new variable with the same name gets created; the syntax used here is just syntactic sugar to keep the code concise. For the full specification of this syntactic sugar, check the grammar in appendix B.

Formally, local operations take the channel as an argument, and then *return* a new channel that contains the next steps of the communication. We can then use shadowing to get the illusion of change. For example, the syntax `c.send[2](n+3); ...` gets translated into the operation `let c=send[2](c,n+3) in ...`. We use this "monadic" shorthand in the examples, since it is more concise, and mimics Haskell's do-notation.

We can now finish the program, adding participant 2 and participant 0 – the main thread:

```
-- Thread that adds 4 to the number
let f2 :: GT|2 -> () =
  \c.
    n = c.receive[1](); -- ?[1]N. ![0]N. End
    c.send[0](n-4);     -- ![0]N. End
    c.close()           -- End

-- Main thread
print
  let c = fork<GT>(c. f1 c, c. f2 c) in
  c.send[1](99);       -- ![1]N. ?[2]N. End
  n = c.receive[2](); -- ?[2]N. End
  c.close();           -- End
  n
```

The program will print `106` onto the screen when run.

Here, the `fork` operation takes a global type, which indicates how communication between threads will flow. It also takes a list of expressions that indicate channels 1 to $n$ respectively. For example, the part `c. f1 c` is the first argument, so is about participant 1. The `c.` part gives a name to the channel for participant 1, and the `f1 c` part is an expression using the channel, returning unit. The channel returned from the fork expression is the one for participant 0, the 'main thread'.

### 2.2.1. A note on linearity in communication

In MPGV, all channels are *linear*. This means that they must be used exactly once in an expression. The linearity therefore ensures that every step of the communication gets used, and gets used only once. For a detailled introduction to linearity, see chapter 3.

The communication operations also must exactly match the current step in the local type of the channel argument. For example, a `send[p]` operation is only valid if used with a channel that is supposed to send something in that step to participant `p`. In other words, the channel's type is of the form `![p]T. LT` where `T` is a type and `LT` is a local type. Likewise, channels in a `receive[p]` operation are of the form `?[p]T. LT`.

If a program gets through the type checker, we can therefore guarantee that the program will not get a runtime error, and will run deadlock-free, since threads execute exactly according to the specified global types.

However, deadlock-free communication does not necessarily imply progress — the notion that communication is guaranteed to continue. Namely, we can still create an infinite loop, for example using the expression `rec f::N->N x::N. f x`. This not only stops the thread, but also the flow of communication with that thread, meaning that some receive operations may become blocked indefinitely.

### 2.2.2. Making a factory

Two of the participants in the previous example look very similar. Just as done in Multiparty GV [12], we can generalise them into a factory function:

```
-- Receive number from thread 0,
-- add n to the number,
-- then send it to thread 1
let add :: N -> ?[0]N. ![1]N. End -> () =
  \n. \c.
    m = c.receive[0](); -- ?[0]N. ![1]N. End
    c.send[1](m+n);     -- ![1]N. End
    c.close()           -- End
```

Here we designate participant 0 as the thread we receive from, and participant 1 as the thread we send to. We also now first take a parameter `n`, which is the amount we will add to the number in the round trip.

However, we now have a problem. We have previously said that the program must match the used types exactly, but this local type doesn't match any of the two we need — we need one that receives from 0 and sends to 2, and one that receives from 1 and sends to 0. For this, the `redirect[p->q]` operation is provided. It tells a channel that, whenever used later, it must use a different participant `q` whenever we encounter a mention of participant `p`.

The idea behind this, is that it doesn't matter what thread runs the steps of a certain participant, as long as it gets the full package. We need to get all the steps, since we later introduce branching communication paths. In that case, we need to know what choice was made, and since we have the full package, we will be notified of the choice made.

To show how redirection works in practice, we give an implementation of the main thread with the new `add` factory:

```
-- Main thread
print
  let c = fork<GT>(
    c. add 3 redirect[1->2](c),       -- ?[0]N. ![2]N. End
    c. add 4 redirect[0->1, 1->0](c) -- ?[1]N. ![0]N. End
  ) in
  c.send[1](99);                      -- ![1]N. ?[2]N. End
  n = c.receive[2]();                 -- ?[2]N. End
  c.close();                          -- End
```

```
10    n
```

Now on line 4, `c` has local type `?[0]N. ![2]N. End`. The redirection returns a new channel, which now mentions participant 1 instead of participant 2. This makes the channel conform exactly with the type of `add`; all communication meant for add-participant 1 now gets correctly sent to the second thread.

The redirection on line 5 is analogous, but requires two redirections. They are written inside of one `redirect` operation, since the participants must switch. If we were to do this in two operations, we would override a participant, which is not what we want.

### 2.2.3. Channel passing

We send any kind of value over channels at any time. This includes channels themselves. To illustrate this feature, we come back to the simple round trip:

```
-- Send a number around in a circle
gtype GT = [0->1]N. [1->2]N. [2->0]N. End
```

In the new scenario — again an example from Multiparty GV [12] — the main thread is lazy, and doesn't want to come up with a number to send around. It only wants to create the other threads, and then receive a result. Therefore, we will make another thread that will perform all necessary communication:

```
-- Delegate main round-trip thread, send back result
gtype GT' = [0->1](![1]N. ?[2]N. End). [1->0]N. End

let g :: GT'|1 -> () =
  \d.
    c = d.receive[0](); -- d: ?[0](GT|0). ![0]N. End
    c.send[1](99);      -- c: ![1]N. ?[2]N. End
    n = c.receive[2](); -- c: ?[2]N. End
    c.close();          -- c: End
    d.send[0](n);       -- d: ![0]N. End
    d.close()           -- d: End
```

The new global type states that we first send the main thread channel, and then send back a result. The new function participates in this communication using channel `d`. It receives the main thread channel as `c`, sends a number on it, receives one, then relays the result back.

Lastly, we write a new main thread:

```
-- Main thread
print
  let c = fork<GT>(c. f1 c, c. f2 c) in
  let d = fork<GT'>(d. g d) in
  d.send[1](c);                          -- ![1](GT|0). ?[1]N. End
  n = d.receive[1]();                    -- ?[1]N. End
  d.close();                             -- End
  n
```

Note that the linearity requirement of `c` is still satisfied. The `send` operation on the main thread consumes the channel, and the corresponding `receive` in `g` reintroduces the channel. This has effectively 'moved' the linearity between threads.

## 2.3. A note on type inference

Note that we have annotated functions with their types. This is necessary, since MPGVR does not have full type inference and thus needs some help deducing certain types. However, not everything needs to be annotated; the bidirectional typing present in the implementation eliminates many annotations.

Bidirectional typing is a strategy where types can be inferred, as long as it can be fully derived from an already annotated or inferred type. So for example, annotating a function with its type will eliminate the need to separately annotate the argument or body. Vice versa, annotating the argument and body eliminates the need to annotate the function. It is recommended to place annotations in top-level let-definitions (not let-in-bindings), since most other types can be derived from those annotations. For the full details of this typing strategy, read chapter 4.

## 2.4. Choice

The last examples introduces algebraic data types and enums. These types can be unpacked using the `match` operation. These allow you to do case-distinction on a type based on a label.

### 2.4.1. Basic examples

We can introduce a "maybe number" using the type `{Just: N; Nothing: ()}`. Now we can make values like `<Just: 2>` or `<Nothing: ()>`.

To use these types, we can prescribe expressions to execute for each label. In the "maybe number" case, it would look like this:

```
match maybeNumber with {
    Just    : n. doSomethingWithTheNumber;
    Nothing: _. doSomethingWithNothing
}
```

The `_` variable name has no special meaning; it is just a convention where we indicate to not care about the value – in the example above, we know it is a unit anyways. Of course, this convention does not apply to linear variables, because they have to be used.

The expressions in all branches must return the same type of value, so that the full match-expression can get that return type.

As a more advanced match-example, we consider weekdays:

```
type WeekDay = {
  Mon:();
  Tue:();
  Wed:();
  Thu:();
  Fri:();
  Sat:();
  Sun:()
}

let nextDay :: WeekDay -> WeekDay =
  \d.
    match d with {
      Mon:_. <Tue:()>;
      Tue:_. <Wed:()>;
      Wed:_. <Thu:()>;
      Thu:_. <Fri:()>;
      Fri:_. <Sat:()>;
      Sat:_. <Sun:()>;
      Sun:_. <Mon:()>
    }
```

We first make an alias `WeekDay` for the weekday type. In this example, none of the weekdays hold any further data, so just contain unit. Then the `nextDay` function will return the weekday after the given one (wrapping around after Sunday).

Note that MPGVR does not support recursive types as of yet. This means that definitions like `type T = ...` can only use type names defined earlier. These type definitions are also not nominal. If two types are the same in terms of content, e.g. `type A = ()` and `type B = ()`, they are seen as the same time. This is in contrast to nominal type systems, where two differently named types are always seen as different. Therefore, the type definitions can better be seen as type "aliases". We leave it to future work to implement recursive and nominal types.

### 2.4.2. Two buyers

We can also have choice in the context of concurrency. In this context, the label doesn't just distinguish between the value sent, but also between the communication steps to execute.

So now we go to a scenario where two buyers want to buy something together. This is an example from Multiparty Asynchronous Session Types [9].

```
-- Seller: 0
-- Buyer A: 1
-- Buyer B: 2
gtype GT = [1->0]N -- Item id
         . [0->1]N -- Price
         . [0->2]N -- Price
         . [1->2]N -- Contribution
         . [2->0]{ -- Can A and B buy together?
             Yes: (). [0->2]N -- Date of delivery
                    . End;
             No : (). End
           }
```

Buyer A tells the seller what they want to buy. The seller then tells both the price. Buyer A puts in some money, and then buyer B decides if they have enough – with that contribution – to buy the item.

The seller is modeled as follows:

```
let seller :: GT|0 -> () =
  \c.
    item = c.receive[1]();
    c.send[1](2 * item); -- Cost is twice the item id in this example
    c.send[2](2 * item); -- Cost is twice the item id in this example
    match c.receive[2]() with {
      Yes: x. let (_, c) = x in
              c.send[2](0); -- Date may vary
              c.close();
      No : x. let (_, c) = x in
              c.close()
    }
```

To keep things simple, we assume the item has a price of twice the id (in euros), and the date has unix timestamp 0.

The `receive` function returns a value-channel pair `(v, c)`, which we unpack after matching. Note that the type of the received channel differs based on which choice we receive, as stated in the global type.

The rest of the program is relatively easy, so we will give it now:

```
let buyerA :: GT|1 -> () =
  \c.
    c.send[0](5); -- Buy item 5
    cost = c.receive[0](); -- Note that cost remains unused
    c.send[2](6); -- A contributes 6 (euros)
    c.close()

let buyerB :: GT|2 -> () =
  \c.
    cost = c.receive[0]();
    contribA = c.receive[1]();
    if contribA + 8 >= cost then -- B contributes 8 (euros)
```

```
        c.send[0](Yes, ()); -- We can buy the item
        date = c.receive[0]();
        c.close()
    else
        c.send[0](No, ()); -- We cannot afford the item
        c.close()

print
  let c = fork<GT>(c. buyerA c, c. buyerB c) in
  seller c
```

Try to follow the execution of the program. It prints unit when it is done.

In this model, buyer A contributes 6 euros, and buyer B is willing to contribute 10 euros. When we run the program, buyer B will therefore say “yes”, since the total contribution of 14 euros is more than the price of 10 euros.

## Consistency

There is one catch to this type of branching: all participants whose behaviour differs in certain branches, must either be the sender or recipient of the label that chooses the branch. This is because they are the only two participants who know about the choice made. Consider the global type from the above example, notably the following part:

```
[2->0]{
    Yes: N . [0->2]N. End;
    No : (). End
}
```

If participant 2 sends a `Yes` value to participant 0, we send the delivery date back. Otherwise, we do nothing. The behaviour for participants 0 and 2 differs based on the choice made, but both are involved with the decision making. Participant 1 is not involved, but its behaviour is consistent; in both cases, participant 1 does nothing.

To show a type which is inconsistent and thus does not type check, we try to send the delivery date to participant 1 as well:

```
[2->0]{
    Yes: N . [0->1]N. [0->2]N. End;
    No : (). End
}
```

the behaviour of particiant 1 is now different in both branches, but participant 1 is not involved in the decision. Always sending a delivery date (possibly some error number when there’s no delivery) to participant 1 fixes the consistency again:

```
[2->0]{
    Yes: N . [0->1]N. [0->2]N. End;
    No : (). [0->1]N. End
}
```

# Chapter 3. Background

In this chapter, we familiarise ourselves with *linearity*, *typing rules*, *type checkers* and *shadowing*. We introduce these concepts my means of the simply-typed lambda calculus. Notation is mainly based on Bidirectional Typing, Multiparty GV and Advanced Topics in Types and Programming Languages [3, 12, 18, respectively]. There are also elements taken from the Haskell and Rust programming languages.

To make definitions and explanations less verbose, we introduce some *variable conventions* [2, 20, 21]. These are implied assumptions on the type of a variable/symbol. For example, any variable using the symbol $e$ is an expression. Of course, depending on the language we are talking about (lambda calculus, MPGV, etc.), the set Expr of all expressions can contain different values; these values are also left implied. If we talk about multiple different expressions, we could annotate the symbols. For example, $e_1$ and $e'$ are still (possibly different) expressions.

In section 3.1, we recall the simply-typed lambda calculus, and introduce typing rules for it. After that, in section 3.2, we introduce linearity — variables must be used exactly once — by introducing a linear variant of the lambda calculus. Then, in section 3.3, we analyse how we can mix linear with non-linear variables, by combining concepts from the previous two lambda calculi into a mixed calculus. Content in these sections is heavily inspired by Advanced Topics in Types and Programming Languages [18].

Section 3.4 introduces shadowing — the act of temporarily overriding a variable with a more "local" value. Here we also describe a formalisation to add shadowing to algorithmic typing judgements. Lastly, in section 3.5, we bidirectionalise the typing rules of the mixed calculus to show how we can remove a lot of type annotations. For this, we make use of a bidirectionalisation recipe described in Bidirectional Typing [3].

We introduce shadowing late, for a good reason. Namely, when checking if an expression is well-typed, accounting for shadowing is non-trivial. It is especially so when combined with linearity, which introduces lifetime constraints. Until then, expressions will only be well-defined if they do not shadow variables. This restriction is also formalised using variable conventions.

Some exercises are given throughout this chapter. They serve as a guide, and are not required to understand this chapter. They are, however, recommended for those not familiar with the material. They can be seen as critical questions on the material, that will not only help with understanding the material, but also make one question the material on a critical level.

## 3.1. Simply-typed lambda calculus

Recall the simply-typed lambda calculus, which we denote with $\lambda$. For now, we use the most minimal definition of the lambda calculus, the syntax for which can be seen below. The syntax is based on the syntax used by Jacobs et al. [12], with some elements taken from Advanced Topics in Types and Programming Languages [18].

$$i, j \in \mathbb{N}, \quad x, y \in \text{Var}$$

$$\begin{aligned} \tau \in \text{Type} &::= \alpha_i \mid \tau \to \tau \\ e \in \text{Expr} &::= x \mid \lambda x_\tau.\ e \mid e\ e \end{aligned}$$

The types $\tau$ are either *type variables* or *single-argument functions* respectively, and the expressions $e$ are either *variables*, *single-argument functions* or *single-argument function application* respectively. Every type variable is distinct, so $\alpha_i = \alpha_j$ if and only if $i = j$. Note that bounded variables are annotated with their type, so that we don't have to perform type inference. This makes defining the typing rules easier.

The syntax is given using a recursive definition. Therefore, we can visualise types and expressions as trees. Then, $\alpha_i$ is a *leaf* type, and the $\tau \to \tau$ is a *branch* type. Likewise, $x$ is a leaf expression, and $\lambda x_\tau.\ e$ and $e\ e$ are branch expressions.

We would like to derive rules, to check if a $\lambda$-expression is well-typed. Part of these well-typed requirements are that variables are within scope. Since there can be free variables in an expression, we must evaluate with respect to an initial context, which should hold bindings for these free variables. We now formally define these contexts.

**Definition 3.1.1.** $\Gamma \in$ Context is a *context*, which is a map of variable-type bindings. Formally, $\Gamma \subseteq \text{Var} \times \text{Type}$, such that for $x \in \text{Var}$ there is at most one $\tau \in \text{Type}$ with $(x, \tau) \in \Gamma$.

We see that a context behaves much like a function. Therefore, we define a sensible operator for this.

**Definition 3.1.2.** The *indexing operator* $\textvisiblespace(\textvisiblespace) : \text{Context} \times \text{Var} \to \text{Type} \cup \{\text{Nothing}\}$ is defined by

$$\Gamma(x) = \begin{cases} \tau & \text{if there is a binding } (x, \tau) \in \Gamma; \\ \text{Nothing} & \text{otherwise.} \end{cases}$$

*Remark.* This operator is well-defined because there is at most one binding per variable. That means we can say:

$$\Gamma(x) = \tau \iff (x, \tau) \in \Gamma.$$

We also need a way to merge two (local) contexts together into a bigger (more global) context. This is governed by the union operator.

**Definition 3.1.3.** We define the (partial function) *context union* $\textvisiblespace \cup \textvisiblespace : \text{Context} \times \text{Context} \to \text{Context}$ as a regular set union. However, contexts with conflicting variable-type bindings may not be joined together, in order for the union to be a closed operation. In other words, $\Gamma_1 \cup \Gamma_2$ is well-defined if and only if there are no $x, \tau_1, \tau_2$ with $\tau_1 \neq \tau_2$ such that $\Gamma_1(x) = \tau_1$ and $\Gamma_2(x) = \tau_2$.

Now we can describe precisely when an expression is considered to be well-typed, and what its type is supposed to be. We do this by looking at each individual expression category separately:

- Given a context $\Gamma$, a (variable) expression $x$ is well-typed with type $\tau$ if and only if $\Gamma(x) = \tau$;
- Given a context $\Gamma$, an expresison $\lambda x_{\tau_1}.\ e$ is well-typed with function type $\tau_1 \to \tau_2$ if and only if $e$ can be evaluated to have the return type $\tau_2$, with respect to the context $\Gamma$ and the extra binding $(x, \tau_1)$;

- Given a context $\Gamma$, an expression $e_1\ e_2$ is well-typed with the return type $\tau_2$ if and only if $e_1$ can be evaluated to have the function type $\tau_1 \to \tau_2$, and $e_2$ can be evaluated to have the argument type $\tau_1$, both with respect to the context $\Gamma$.

Below, we show some examples of expressions and whether they are well-typed according to these constraints. Parentheses are given for clarity on the order of operations; they serve no further purpose.

| **Expression** | **Well-typed** |
|---|---|
| $\lambda x_{\alpha_0}.\ x$ | yes |
| $\lambda x_{\alpha_0 \to \alpha_1}.\ (x\ y)$ | if $\Gamma(y) = \alpha_0$ |
| $\lambda x_{\alpha_0}.\ (x\ y)$ | no: $x$ is not a function |
| $(\lambda x_{\alpha_0}.\ x)\ (\lambda y_{\alpha_1}.\ y)$ | no: argument type mismatch |
| $\lambda x_{\alpha_0}.\ (\lambda x_{\alpha_0}.\ x)$ | not well-defined: shadows $x$ |

**Exercise 3.1.4.** (a) Verify that $(\lambda x_{\alpha_0 \to \alpha_0}.\ x\ y)\ (\lambda x_{\alpha_0}.\ x)$ is well-typed with respect to the context $\{(y, \alpha_0)\}$.

(b) Explain why $(\lambda x_{\alpha_0 \to \alpha_1}.\ x\ y)\ (\lambda x_{\alpha_0}.\ x)$ is not well-typed, regardless of context.

### 3.1.1. Typing rules

We have already stated when an expression is considered to be well-typed, but we can formalise this using *typing rules*. This formalisation allows us to reason more rigorously about typing of expressions, and can make it more clear how to translate the rules into a type checking program. To start this off, we define the *typing judgement*.

**Definition 3.1.5.** A *typing judgement* $\Gamma \vdash_\lambda e : \tau$ is a predicate that states whether a $\lambda$-expression $e$ can be evaluated to have type $\tau$ with respect to the context $\Gamma$, only by using the corresponding typing rules as valid deductions.

*Remark.* This typing judgement is also defined for other languages and calculi, not just for $\lambda$. The language used in the judgement is denoted with a subscript. For example, the language $\lambda^+$ (defined later) uses the judgement $\Gamma \vdash_{\lambda^+} e : \tau$.

The notation may remind one of the logical notation $T \vdash \varphi$, which states if the formula $\varphi$ can be proven to be true, only by using the assumptions in the theory $T$ and the logical deduction rules. Typing rules are consistent with logical notation, in that we also use deduction rule notation to define the formal typing rules. The notation (and related notation about e.g. contexts) is used in many other works [3, 12, 18]. This is what the rules look like:

$$\frac{}{\Gamma \cup \{(x, \tau)\} \vdash_\lambda x : \tau}\ (\text{var}_\lambda)$$

$$\frac{\Gamma \cup \{(x, \tau_1)\} \vdash_\lambda e : \tau_2}{\Gamma \vdash_\lambda \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2}\ (\text{func}_\lambda)$$

$$\frac{\Gamma \vdash_\lambda e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_\lambda e_2 : \tau_1}{\Gamma \vdash_\lambda e_1\ e_2 : \tau_2}\ (\text{app}_\lambda)$$

To read these typing rules, first look at the bottom. There we see the form of the expression that we wish to derive the type of, and we see the form that the context must have. Then, look at the top to see further requirements that we must satisfy. Note that in these deduction rules a lot of assumptions are made on the forms of variables. For example, we may only use the $\text{app}_\lambda$ rule if $e_1$ yields specifically a function type. In the end, if the rule we want to use is well-defined, the type in the bottom will flow right out.

As you can see, the three rules correspond nicely with the three different expression forms from the syntax (variables, functions and function application). Therefore, we can make proof trees that follows the form of an expression, which shows that its typing judgement holds. So we can call the rule on variables a *leaf* rule, and the other two rules *branch* rules.

**Exercise 3.1.6.** Compare the formal typing rules with the informal description given earlier.

**Exercise 3.1.7.** Draw a proof-tree that deduces the type of the expression in exercise 3.1.4(a).

Note that the the rule $\text{func}_\lambda$ has an implicit assumption: that $\Gamma(x) = \text{Nothing}$. This is to satisfy the variable convention, namely that $x$ is a new variable with respect to the scope. This convention is also implicit in the previous examples.

Also note that the previous exercise implies that there is at most one type $\tau$ for which a typing judgement $\Gamma \vdash_\lambda e : \tau$ holds. In other words, the typing rules allow us to deduce unique types for expressions. We prove this fact later in the section.

## 3.1.2. Extending the calculus

The lambda calculus $\lambda$ is a very minimal calculus. It is not well-suited for practical applications. This could make the typing rules still feel a bit abstract. Therefore, to get more comfortable with them, we now consider a more involved and practical example. Below, we show an extended version of $\lambda$, which we call $\lambda^+$. This calculus adds support for natural numbers, pairs and if-then-else statements. In if-then-else statements, the true-branch is taken whenever we get a positive integer, and the false-branch is taken whenever we get 0.

$$i, j, n \in \mathbb{N}, \quad x, y \in \text{Var}$$

$$\begin{aligned} \tau \in \text{Type} &::= \alpha_i \mid \tau \to \tau \mid \mathbf{N} \mid \tau \times \tau \\ e \in \text{Expr} &::= x \mid \lambda x_\tau .\ e \mid e\ e \mid n \mid (e, e) \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \end{aligned}$$

Before we give the extension of the typing rules for this new calculus, we first note one important requirement: the result type in both branches of an if-then-else statement must be the same. Otherwise, we cannot give the if-then-else statement a proper result type.

Keeping this constraint in mind, we give the new typing rules. These extend the typing rules in 3.1.1, which remain the same in $\lambda^+$.

$$\frac{}{\Gamma \vdash_{\lambda^+} n : \mathbf{N}}\ (\text{nat}_{\lambda^+})$$

$$\frac{\Gamma \vdash_{\lambda^+} e_1 : \tau_1 \qquad \Gamma \vdash_{\lambda^+} e_2 : \tau_2}{\Gamma \vdash_{\lambda^+} (e_1, e_2) : \tau_1 \times \tau_2}\ (\text{pair}_{\lambda^+})$$

$$\frac{\Gamma \vdash_{\lambda^+} e_1 : \mathbf{N} \qquad \Gamma \vdash_{\lambda^+} e_2 : \tau \qquad \Gamma \vdash_{\lambda^+} e_3 : \tau}{\Gamma \vdash_{\lambda^+} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (\text{if-then-else}_{\lambda^+})$$

**Exercise 3.1.8.** Describe these new typing rules informally, but precisely.

**Exercise 3.1.9.** Add the arithmetic operations $+$ and $-$ to the calculus, by adding to the existing syntax and typing rules.

## 3.1.3. Type checker

With a good grasp of the typing rules, we return to the minimal calculus $\lambda$. We wish to create an algorithmic type checker, that follows the typing rules in 3.1.1. This type checker can be written in a programming language, like Haskell. However, for purposes of this document, we define one using pure mathematical functions, since we are all familiar with those.

Instead of seeing the typing judgement as a predicate, we can see it as a function on the expression and context, returning either a type if we can derive one, or Error if we cannot. This interpretation will form the basis of our type checker.

**Definition 3.1.10.** The function $\text{check}_\lambda : \text{Expr} \times \text{Context} \to \text{Type} \cup \{\text{Error}\}$ is a *type checking function*; in other words, $\text{check}_\lambda(e, \Gamma) = \tau$ if we give the return type $\tau$ to the expression $e$, with respect to the context $\Gamma$, and $\text{check}_\lambda(e, \Gamma) = \text{Error}$ if we declare the expression to not be well-typed, again with respect to the context $\Gamma$.

*Remark.* The type checking function, just like the typing judgement, is not limited to just $\lambda$. The language is a parameter to the function.

A way of implementing $\text{check}_\lambda$ is given below. Note that this function is well-defined; it does not have multiple solutions for a given input. This is because the function is defined inductively on the construction of the expression.

$$\begin{aligned}
\text{check}_\lambda(x, \Gamma) &= \Gamma(x) \\
\text{check}_\lambda(\lambda x_{\tau_1}.\ e, \Gamma) &= \begin{cases} \tau_1 \to \tau_2 & \text{if } \text{check}(e, \Gamma \cup \{(x, \tau_1)\}) = \tau_2; \\ \text{Error} & \text{otherwise.} \end{cases} \\
\text{check}_\lambda(e_1\ e_2, \Gamma) &= \begin{cases} \tau_2 & \text{if } \text{check}_\lambda(e_1, \Gamma) = \tau_1 \to \tau_2 \text{ and } \text{check}_\lambda(e_2, \Gamma) = \tau_1; \\ \text{Error} & \text{otherwise.} \end{cases}
\end{aligned}$$

The goal of this implementation is to coincide with the typing judgements. This is proven in the following theorem.

**Theorem 3.1.11** (check implements the rules of $\lambda$)**.** We have the equivalence

$$\text{check}_\lambda(e, \Gamma) = \tau \iff \Gamma \vdash_\lambda e : \tau.$$

**Corollary 3.1.12** (check only accepts $\lambda$)**.** The above theorem implies that

$$\text{check}_\lambda(e, \Gamma) = \text{Error} \iff \neg(\Gamma \vdash_\lambda e : \tau) \text{ for any } \tau,$$

or in other words, that the type checking function and the typing rules have the same idea of what is well-typed and what is not.

**Corollary 3.1.13** (type uniqueness)**.** Another observation we can draw from this theorem is that a typing judgement $\Gamma \vdash_\lambda e : \tau$ for an expression $e$ and context $\Gamma$ holds for at most one type. Therefore, in both the type checker and typing rules, we can talk about *the* type of an expression.

*Proof.* We prove the theorem for each expression category individually, and use induction on the construction of the expression.

- Firstly, the base case is for variable-expressions $x$.

  ($\Longrightarrow$) Assume that $\text{check}_\lambda(x, \Gamma) = \tau$. Then $\Gamma(x) = \tau$, so there is a binding $(x, \tau) \in \Gamma$. Therefore, the typing rule $\text{var}_\lambda$ satisfies its singular assumption, because $\Gamma = \Gamma \cup \{(x, \tau)\}$, such that $\Gamma \vdash_\lambda x : \tau$.

  ($\Longleftarrow$) Assume that $\Gamma \vdash_\lambda x : \tau$. In the typing rule $\text{var}_\lambda$, we see that there is one assumption made. This is that the context must contain a binding for $x$, specifically the binding $(x, \tau)$. This means that $\Gamma(x) = \tau$, by definition of the indexing operator. Therefore, $\text{check}_\lambda(x, \Gamma) = \tau$.

- We now prove the theorem for function-expressions $\lambda x_{\tau_1}.\ e$. For this, we have the induction hypothesis $\text{check}_\lambda(e, \Gamma) = \tau \iff \Gamma \vdash_\lambda e : \tau$ for all contexts $\Gamma$ and types $\tau$.

  ($\Longrightarrow$) Assume that $\text{check}(\lambda x_{\tau_1}.\ e, \Gamma) = \tau$. Then we may write $\tau = \tau_1 \to \tau_2$, by definition. Again by definition, we see that $\text{check}_\lambda(e, \Gamma \cup \{(x, \tau_1)\}) = \tau_2$ holds. From the induction hypothesis, it now follows that $\Gamma \cup \{(x, \tau_1)\} \vdash_\lambda e : \tau_2$. It is now clear that the rule $\text{func}_\lambda$ gives us that $\Gamma \vdash \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2$.

($\Longleftarrow$) Assume that $\Gamma \vdash \lambda x_{\tau_1}.\ e : \tau$. Then we may write $\tau = \tau_1 \to \tau_2$, by definition of the rule $\text{func}_\lambda$. Again by definition, we see that $\Gamma \cup \{(x, \tau_1)\} \vdash_\lambda e : \tau_2$ holds. From the induction hypothesis, it now follows that $\text{check}_\lambda\,(e, \Gamma \cup \{(x, \tau_1)\}) = \tau_2$. It is now clear that $\text{check}\,(\lambda x_{\tau_1}.\ e, \Gamma) = \tau_1 \to \tau_2$.

- Lastly, we prove the theorem for application-expressions $e_1\ e_2$. For this, we have the induction hypotheses $\text{check}_\lambda\,(e_1, \Gamma) = \tau \iff \Gamma \vdash_\lambda e_1 : \tau$, and $\text{check}_\lambda\,(e_2, \Gamma) = \tau \iff \Gamma \vdash_\lambda e_2 : \tau$ for all contexts $\Gamma$ and types $\tau$.

  ($\Longrightarrow$) Assume that $\text{check}\,(e_1\ e_2, \Gamma) = \tau_2$. Then $\text{check}_\lambda\,(e_1, \Gamma) = \tau_1 \to \tau_2$ and $\text{check}_\lambda\,(e_2, \Gamma) = \tau_1$ hold by definition. From the induction hypotheses, it now follows that $\Gamma \vdash_\lambda e_1 : \tau_1 \to \tau_2$ and that $\Gamma \vdash_\lambda e_2 : \tau_1$. It is now clear that the rule $\text{app}_\lambda$ gives us that $\Gamma \vdash_\lambda e_1\ e_2 : \tau_2$.

  ($\Longleftarrow$) Assume that $\Gamma \vdash_\lambda e_1\ e_2 : \tau_2$. By definition of the rule $\text{app}_\lambda$, we see that $\Gamma \vdash_\lambda e_1 : \tau_1 \to \tau_2$ and that $\Gamma \vdash_\lambda e_2 : \tau_1$ hold. From the induction hypotheses, it now follows that $\text{check}_\lambda\,(e_1, \Gamma) = \tau_1 \to \tau_2$ and $\text{check}_\lambda\,(e_2, \Gamma) = \tau_1$. It is now clear that $\text{check}\,(e_1\ e_2, \Gamma) = \tau_2$.

□

**Exercise 3.1.14.** Extend the type checking function to $\lambda^+$. Verify (no formal proof needed) that an analog of the theorem holds in $\lambda^+$ as well.

## 3.2. Linear lambda calculus

Now that we are familiar with typing rules and type checkers, we indroduce *linearity*; it is a requirement on a variable that it must be used exactly once in an expression. When a variable has the linearity requirement, we call it a *linear* variable. Variables that don't have this requirement are called *unrestricted*, or *copy*. The term "copy" is taken from the `Copy`[1] trait in the Rust programing language.

We explain linearity again by means of a lambda calculus. In this case, we present a variant of the lambda calculus that is fully linear, which we call $\lambda^{\mathbb{1}}$. Apart from notation, the syntax is the same as with $\lambda$. The $\mathbb{1}$ is taken from Haskell's linear function extension[2]. Like in the previous section, we start by showing this new notation. The notational change isn't strictly necessary for this section, but makes it easier to draw parallels with a mixed lambda calculus later on.

$$i, j \in \mathbb{N}, \quad x, y \in \text{Var}$$

$$\begin{aligned} \tau \in \text{Type} &::= \alpha_i^{\mathbb{1}} \mid \tau \multimap \tau \\ e \in \text{Expr} &::= x \mid \lambda^{\mathbb{1}} x_\tau.\ e \mid e\ e \end{aligned}$$

An expression in $\lambda^{\mathbb{1}}$ still follows the typing rules of $\lambda$, but adds the linearity requirement to all (sub)expressions. We illustrate this new requirement with some examples:

| **Expression** | **Well-typed** |
|---|---|
| $x$ | if $\Gamma(x) \neq$ Nothing |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}}}.\ x$ | yes |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}} \multimap \alpha_0^{\mathbb{1}}}.\ (\lambda^{\mathbb{1}} y_{\alpha_0^{\mathbb{1}}}.\ x\ y)$ | yes |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}}}.\ (\lambda^{\mathbb{1}} y_{\alpha_0^{\mathbb{1}}}.\ x)$ | no: $y$ is unused |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}} \multimap (\alpha_0^{\mathbb{1}} \multimap \alpha_0^{\mathbb{1}})}.\ (\lambda^{\mathbb{1}} y_{\alpha_0^{\mathbb{1}}}.\ (x\ y)\ y)$ | no: $y$ is used twice |

### 3.2.1. Regular typing rules

Before we give typing rules for this calculus, we need a strategy for how we ensure that variables satisfy the linearity requirement. We split this strategy into two steps:

(a) ensure that no variables are unused;

(b) ensure that variables cannot be used more than once.

If we implement both strategies into the typing rules, we have implemented linearity. To start off, we implement the first strategy by means of a predicate.

**Definition 3.2.1.** On contexts in linear systems, we define a notion of *disjointness*. We say $\Gamma_1 \perp \Gamma_2$, i.e. that the contexts are *disjoint*, if and only if no variable has a binding in both contexts. In other words, $\Gamma_1 \perp \Gamma_2$ whenever $\Gamma_1(x) \neq \text{Nothing} \implies \Gamma_2(x) = \text{Nothing}$ for all variables $x$. We also write $\Gamma = \Gamma_1 \perp \Gamma_2$ to indicate that $\Gamma$ is a disjoint partition of $\Gamma_1$ and $\Gamma_2$, i.e. $\Gamma = \Gamma_1 \cup \Gamma_2$ and $\Gamma_1 \perp \Gamma_2$.

*Remark.* When we introduce the mixed lambda calculus later on, we need a slightly different notion of disjointness. In that system, contexts are only disjoint with respect to linear variables in them, not with respect to copy variables.

**Corollary 3.2.2.** By definition, $\Gamma_1 \perp \Gamma_2$ implies that the operation $\Gamma_1 \cup \Gamma_2$ is well-defined. From $\Gamma_1 \perp \Gamma_2$ it also follows that $(x, \tau) \in \Gamma_1 \cup \Gamma_2$ implies exactly one of $(x, \tau) \in \Gamma_1$ or $(x, \tau) \in \Gamma_2$.

Using this notion of disjointness, we can implement the typing rules for $\lambda^{\mathbb{1}}$:

[1] https://doc.rust-lang.org/std/marker/trait.Copy.html
[2] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/linear_types.html

$$\frac{}{\{(x,\tau)\} \vdash_{\mathbb{1}} x : \tau}\ (\text{var}_{\mathbb{1}}^{\text{reg}})$$

$$\frac{\Gamma \cup \{(x,\tau_1)\} \vdash_{\mathbb{1}} e : \tau_2}{\Gamma \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2}\ (\text{func}_{\mathbb{1}}^{\text{reg}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2}\ (\text{app}_{\mathbb{1}}^{\text{reg}})$$

Note that for readability, we shorted the language name $\lambda^{\mathbb{1}}$ down to just $\mathbb{1}$. The $\text{var}_{\mathbb{1}}^{\text{reg}}$ rule is different from the $\text{var}_{\lambda}$ rule, in that this new rule requires the context to consist of only one variable. This is to ensure that no variables are left unused; if there were another variable inside of the context, it won't be used up, since $\text{var}_{\mathbb{1}}^{\text{reg}}$ is a leaf rule that only consumes one variable. Since there are no other leaf rules, this ensures property (a), that no variables are unused.

The other change is made in the $\text{app}_{\mathbb{1}}^{\text{reg}}$. This is the only branch rule that splits the expression into more than one subexpression, so it introduces the opportunity for variables to be put inside both subexpressions. The requirement that the context be partitioned into two *disjoint* parts, ensures that each variable moves to exactly one subexpression (see Corollary 3.2.2). Since this is the only branch rule that splits an expression into more than one subexpression, property (b) — that no variables are used more than once — is also satisfied.

Since these two changes are just restrictions added onto the typing rules for $\lambda$, one could consider $\lambda^{\mathbb{1}}$ a sort-of "subset" of $\lambda$, since every well-typed expression in $\lambda^{\mathbb{1}}$ is also well-typed in $\lambda$. We will talk in-depth about this observation in the next section.

**Exercise 3.2.3.** (a) Give a proof tree for the third example, showing that the type of the expression is $(\alpha_0^{\mathbb{1}} \multimap \alpha_0^{\mathbb{1}}) \multimap \alpha_0^{\mathbb{1}}$.

(b) Show that the other examples give the correct result with the formal typing rules (i.e. whether they're considered well-typed or not).

### 3.2.2. Algorithmic typing rules

As with $\lambda$, we wish to create a type checking algorithm, that can compute types of expressions. However, we realise that there is one issue when translating the typing rules into an algorithm: how do we split the context into two? Which variables should go where?

One way to solve this problem is to try all possible combinations of split contexts, and take whatever works (and error if no combination works). However, this is error-prone and inefficient. The route that we take, is to translate these *regular* typing rules into *algorithmic* typing rules, and then derive a type checker from these new rules.

The algorithmic typing rules have no notion of *splitting* contexts, and instead opts to *weave*[3] one context through the type derivation, adding to and removing variables from it throughout the process. This leaves a (modified) context as a result. Figure 3.1 schematically shows the difference between the two notions. This weaving notion removes the need to guess, making the process less error-prone and more efficient.

Note that the algorithmic process has a resulting context. This means that the process does not check if variables from the initial context have actually been used up. This linearity check has to be done separately if needed.

**Definition 3.2.4.** An *algorithmic typing judgement* $\Gamma_1 \vdash_{\lambda} e : \tau \dashv \Gamma_2$ is a predicate that states whether a $\lambda$-expression can be evaluated to have type $\tau$ with respect to the context $\Gamma_1$, leaving $\Gamma_2$ as the resulting context, only by using the corresponding typing rules as valid deductions.

[3]Advanced Topics in Types and Programming Languages describes this process using *input* and *output* contexts [18] instead.

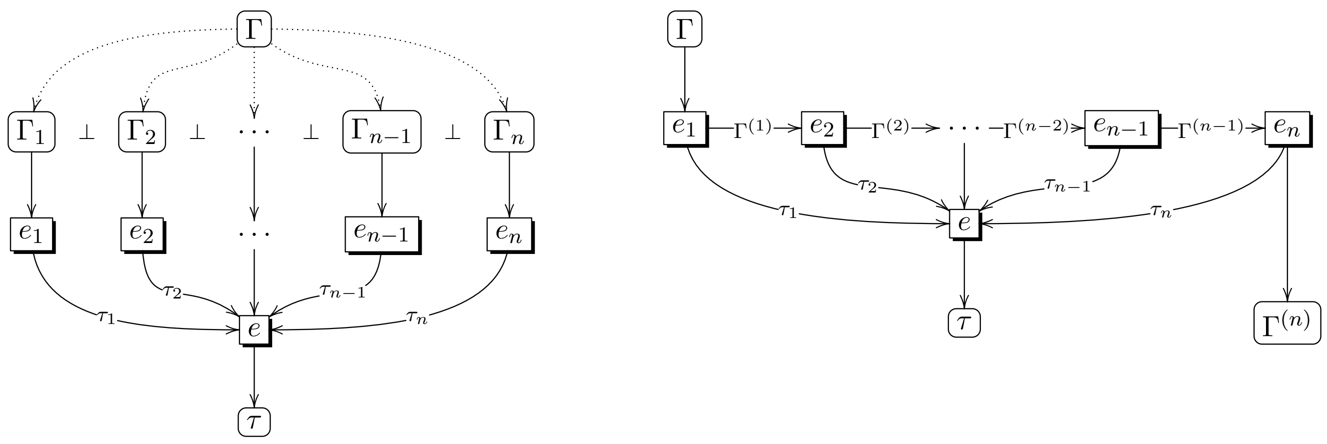
$\Gamma$
$\Gamma_1$ $\perp$ $\Gamma_2$ $\perp$ $\cdots$ $\perp$ $\Gamma_{n-1}$ $\perp$ $\Gamma_n$
$e_1$ $e_2$ $\cdots$ $e_{n-1}$ $e_n$
$\tau_1$ $\tau_2$ $\tau_{n-1}$ $\tau_n$
$e$
$\tau$
$\Gamma$
$e_1$ $\Gamma^{(1)}$ $e_2$ $\Gamma^{(2)}$ $\cdots$ $\Gamma^{(n-2)}$ $e_{n-1}$ $\Gamma^{(n-1)}$ $e_n$
$\tau_1$ $\tau_2$ $\tau_{n-1}$ $\tau_n$
$e$
$\tau$
$\Gamma^{(n)}$

Figure 3.1: Type checking by means of splitting contexts (left) versus type checking by means of weaving contexts (right).

**Definition 3.2.5.** Let $\Gamma - x$ denote the context in which variable $x$ was removed, i.e.

$$\Gamma - x := \begin{cases} \Gamma \setminus \{(x, \tau)\} & \text{if } \Gamma(x) = \tau; \\ \Gamma & \text{if } \Gamma(x) = \text{Nothing}. \end{cases}$$

Below, we show the algorithmic typing rules for $\lambda^{\mathbb{1}}$. We note the important differences:

- In the rule $\text{var}_{\mathbb{1}}^{\text{alg}}$, we check if a type for the variable exists in the context (as usual), and then assign that type if it exists. The resulting context is one where the binding is removed, since we may not use the same variable more than once.
- In the rule $\text{func}_{\mathbb{1}}^{\text{alg}}$, the resulting context is the one from the subexpression. However, this context may not have the variable $x$ anymore, since it must be used in the subexpression $e$.
- In the rule $\text{app}_{\mathbb{1}}^{\text{alg}}$, we see how the context $\Gamma_1$ "weaves" through the subexpressions via $\Gamma_2$ into $\Gamma_3$. This ensures that no variable is used more than once; if $e_1$ uses a variable, it will be inaccessible to $e_2$, since it's been removed once we get to $\Gamma_2$.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathbb{1}} x : \tau \dashv \Gamma - x} \ (\text{var}_{\mathbb{1}}^{\text{alg}})$$

$$\frac{\Gamma_1 \cup \{(x, \tau_1)\} \vdash_{\mathbb{1}} e : \tau_2 \dashv \Gamma_2 \qquad \Gamma_2(x) = \text{Nothing}}{\Gamma_1 \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2 \dashv \Gamma_2} \ (\text{func}_{\mathbb{1}}^{\text{alg}})$$

$$\frac{\Gamma_1 \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2 \dashv \Gamma_3} \ (\text{app}_{\text{M}}^{\text{alg}})$$

**Exercise 3.2.6.** Using the algorithmic rules, give a proof tree for the third example in **??**, showing that the type of the expression is $(\alpha_0^{\mathbb{1}} \multimap \alpha_0^{\mathbb{1}}) \multimap \alpha_0^{\mathbb{1}}$. Use the context $\{(a, \alpha_1^{\mathbb{1}})\}$ as the initial context. Show that the resulting context is then "unchanged", i.e. still $\{(a, \alpha_1^{\mathbb{1}})\}$.

From the typing rules, we make an interesting observation: we only "consume" variables from the context in the $\text{var}_{\mathbb{1}}^{\text{reg}}$ rule. Therefore, if there is a variable $x$ in the initial context, and the expression does not have this variable, the variable will remain in the resulting context. We formalise this observation in the following lemma:

**Lemma 3.2.7** (unused variables are not removed)**.** For a variable $x$ and an expression $e$ not using $x$, and for all types $\tau_x$, we have that

$$\Gamma_1 \vdash_{\mathbb{1}} e : \tau \dashv \Gamma_2 \iff \Gamma_1 \cup (x, \tau_x) \vdash_{\mathbb{1}} e : \tau \dashv \Gamma_2 \cup (x, \tau_x).$$

We now have left to show that the algorithmic typing rules still have the same idea of what is well-typed. To do this, we prove the following theorem:

**Theorem 3.2.8.** The regular and algorithmic typing rules for $\lambda^{\mathbb{1}}$ have the same notion of well-typing. More formally,

$$\Gamma \vdash_{\mathbb{1}} e : \tau \iff \Gamma \vdash_{\mathbb{1}} e : \tau \dashv \emptyset.$$

*Proof.* We use induction on the construction of the expression $e$:

- Firstly, the base case is for variable-expressions $x$.

  ($\Longrightarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} x : \tau$. Then, we know that $\Gamma$ is of the form $\{(x, \tau)\}$ using rule $\text{var}_{\mathbb{1}}^{\text{reg}}$. Therefore, $\Gamma \setminus (x, \tau) = \emptyset$, so rule $\text{var}_{\mathbb{1}}^{\text{alg}}$ implies that $\Gamma \vdash_{\mathbb{1}} x : \tau \dashv \emptyset$ holds.

  ($\Longleftarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} x : \tau \dashv \emptyset$. We know that rule $\text{var}_{\mathbb{1}}^{\text{alg}}$ holds, and since $\Gamma \setminus (x, \tau) = \emptyset$ only when $\Gamma = \{(x, \tau)\}$, rule $\text{var}_{\mathbb{1}}^{\text{reg}}$ holds such that $\Gamma \vdash_{\lambda} x : \tau$ holds.

- We now prove the theorem for function-expressions $\lambda^{\mathbb{1}} x_{\tau_1}.\, e$. For this, we have the induction hypothesis $\Gamma \vdash_{\mathbb{1}} e : \tau \iff \Gamma \vdash_{\mathbb{1}} e : \tau \dashv \emptyset$ for all contexts $\Gamma$ and types $\tau$.

  ($\Longrightarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\, e : \tau_1 \multimap \tau_2$. Using rule $\text{func}_{\mathbb{1}}^{\text{reg}}$, we know that $\Gamma \cup (x, \tau_1) \vdash_{\mathbb{1}} e : \tau_2$. The induction hypothesis then lets us conclude $\Gamma \cup (x, \tau_1) \vdash_{\mathbb{1}} e : \tau_2 \dashv \emptyset$. Since the resulting context is empty, is notably doesn't contain $x$. Therefore, $\text{func}_{\mathbb{1}}^{\text{alg}}$ implies that $\Gamma \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\, e : \tau_1 \multimap \tau_2 \dashv \emptyset$ holds.

  ($\Longleftarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\, e : \tau_1 \multimap \tau_2 \dashv \emptyset$. Using rule $\text{func}_{\mathbb{1}}^{\text{alg}}$, we know that $\Gamma \cup (x, \tau_1) \vdash_{\mathbb{1}} e : \tau_2 \dashv \emptyset$. The induction hypothesis then lets us conclude $\Gamma \cup (x, \tau_1) \vdash_{\mathbb{1}} e : \tau_2$. Therefore, $\text{func}_{\mathbb{1}}^{\text{reg}}$ implies that $\Gamma \vdash_{\mathbb{1}} \lambda^{\mathbb{1}} x_{\tau_1}.\, e : \tau_1 \multimap \tau_2$ holds.

- Lastly, we prove the theorem for application-expressions $e_1\ e_2$. For this, we have the induction hypotheses $\Gamma \vdash_{\mathbb{1}} e_1 : \tau \iff \Gamma \vdash_{\mathbb{1}} e_1 : \tau \dashv \emptyset$ and $\Gamma \vdash_{\mathbb{1}} e_2 : \tau \iff \Gamma \vdash_{\mathbb{1}} e_2 : \tau \dashv \emptyset$ for all contexts $\Gamma$ and types $\tau$.

  ($\Longrightarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2$. Using rule $\text{app}_{\mathbb{1}}^{\text{reg}}$, we know that $\Gamma = \Gamma_1 \perp \Gamma_3$ such that $\Gamma_1 \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2$ and $\Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1$. The first induction hypothesis and lemma Theorem 3.2.7 then let us first conclude $\Gamma \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2 \dashv \Gamma_2$. The second induction hypothesis then implies that $\Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1 \dashv \emptyset$. Therefore, $\text{func}_{\mathbb{1}}^{\text{alg}}$ implies that $\Gamma \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2 \dashv \emptyset$ holds.

  ($\Longleftarrow$) Assume that $\Gamma \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2 \dashv \emptyset$. Using rule $\text{app}_{\mathbb{1}}^{\text{alg}}$, we know that there is a $\Gamma_2$ such that $\Gamma \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2 \dashv \Gamma_2$ and $\Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1 \dashv \emptyset$. Let $\Gamma_1$ be the context such that $\Gamma = \Gamma_1 \perp \Gamma_2$. The induction hypotheses and lemma Theorem 3.2.7 then let us conclude $\Gamma_1 \vdash_{\mathbb{1}} e_1 : \tau_1 \multimap \tau_2$. The second induction hypothesis then implies that $\Gamma_2 \vdash_{\mathbb{1}} e_2 : \tau_1$. Therefore, $\text{func}_{\mathbb{1}}^{\text{reg}}$ implies that $\Gamma \vdash_{\mathbb{1}} e_1\ e_2 : \tau_2$ holds.

□

### 3.2.3. Type checker

Now that we have the algorithmic typing rules, creating a type checking function is rather uneventful. The one major change we need to make, is to the definition of the type checking function, to support context weaving.

**Definition 3.2.9.** The function $\text{check}_{\lambda} : \text{Expr} \times \text{Context} \to (\text{Type} \times \text{Context}) \cup \{\text{Error}\}$ is an *algorithmic type checking function* for the given language $\lambda$; in other words, $\text{check}_{\lambda}(e, \Gamma_1) = (\tau, \Gamma_2)$ if we give the return type $\tau$ and resulting context $\Gamma_2$ to the expression $e$, with respect to the initial context $\Gamma_1$. We say that $\text{check}(e, \Gamma_1) = \text{Error}$ if we declare the expression to not be well-typed, again with respect to the initial context $\Gamma_1$.

With this definition, we show the type checking function for $\lambda^{\mathbb{1}}$:

$$\text{check}_{\mathbb{1}}\,(x, \Gamma) = \begin{cases} (\tau, \Gamma \setminus (x, \tau)) & \text{if } \Gamma(x) = \tau; \\ \text{Error} & \text{otherwise.} \end{cases}$$

$$\text{check}_{\mathbb{1}}\,(\lambda^{\mathbb{1}} x_{\tau_1}.\ e, \Gamma_1) = \begin{cases} (\tau_1 \multimap \tau_2, \Gamma_2) & \text{if } \text{check}_{\mathbb{1}}\,(e, \Gamma_1 \cup \{(x, \tau_1)\}) = (\tau_2, \Gamma_2) \text{ and } \Gamma_2(x) = \text{Nothing}; \\ \text{Error} & \text{otherwise.} \end{cases}$$

$$\text{check}_{\mathbb{1}}\,(e_1\ e_2, \Gamma_1) = \begin{cases} (\tau_2, \Gamma_3) & \text{if } \text{check}_{\mathbb{1}}\,(e_1, \Gamma_1) = (\tau_1 \multimap \tau_2, \Gamma_2) \text{ and } \text{check}_{\mathbb{1}}\,(e_2, \Gamma_2) = (\tau_1, \Gamma_3); \\ \text{Error} & \text{otherwise.} \end{cases}$$

It is easy to see that this type checking function is "analogous" to the algorithmic typing rules. For the sake of completeness, we will give the formal theorem. However, since we are already familiar with what a proof to such a theorem looks like, and such proof is rather tedious, we leave out the proof; there are enough parallels between the typing rules and the type checking function that it is already quite evident. Also note that we have analogous corollaries to this theorem as in the previous section.

**Theorem 3.2.10** (check implements the rules of $\lambda^{\mathbb{1}}$)**.** We have the equivalence

$$\text{check}_{\mathbb{1}}\,(e, \Gamma_1) = (\tau, \Gamma_2) \iff \Gamma_1 \vdash_{\mathbb{1}} e : \tau \dashv \Gamma_2.$$

## 3.3. Mixed lambda calculus

The last language we introduce in this part is the mixed lambda calculus $\lambda^{\mathsf{b}}$. (The "b" is supposed to be read like a "linearity boolean".) It combines both the copy constructions from $\lambda$ and linear constructions from $\lambda^{\mathbb{1}}$. We give the syntax for this language, and some examples. It is precisely the combination of the previous languages' syntaxes.

$$i, j \in \mathbb{N}, \quad x, y \in \text{Var}$$

$$\begin{aligned} \tau &::= \alpha_i \mid \alpha_i^{\mathbb{1}} \mid \tau \to \tau \mid \tau \multimap \tau \\ e &::= x \mid \lambda x_\tau.\ e \mid \lambda^{\mathbb{1}} x_\tau.\ e \mid e\ e \end{aligned}$$

| **Expression** | **Well-typed** |
|---|---|
| $x$ | if $\Gamma(x) \neq$ Nothing |
| $\lambda x_{\alpha_0}.\ x$ | yes |
| $\lambda x_{\alpha_0 \to \alpha_1}.\ (x\ y)$ | if $\Gamma(y) = \alpha_0$ |
| $\lambda^{\mathbb{1}} x_{\alpha_0}.\ (x\ y)$ | no: $x$ is not a function |
| $(\lambda x_{\alpha_0}.\ x)\ (\lambda y_{\alpha_1}.\ y)$ | no: argument type mismatch |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}}}.\ x$ | yes |
| $\lambda x_{\alpha_0 \multimap \alpha_0}.\ (\lambda^{\mathbb{1}} y_{\alpha_0}.\ x\ y)$ | yes |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}}}.\ (\lambda y_{\alpha_0^{\mathbb{1}}}.\ x)$ | no: $y$ is unused |
| $\lambda^{\mathbb{1}} x_{\alpha_0^{\mathbb{1}}}.\ (\lambda y_{\alpha_0}.\ x)$ | yes |
| $\lambda x_{\alpha_0^{\mathbb{1}} \to (\alpha_0^{\mathbb{1}} \to \alpha_0^{\mathbb{1}})}.\ (\lambda y_{\alpha_0^{\mathbb{1}}}.\ (x\ y)\ y)$ | no: $y$ is used twice |

The two kinds of types from $\lambda$ are the copy types, and those from $\lambda^{\mathbb{1}}$ are the linear types.

A function expression may use more (free) variables than just its argument variable. If such a variable is linear, the function expression must be linear as well. This is to prevent the variable being used more than once, if the function is called more than once. The argument variable being linear does not induce any problems with copy functions. This is because on every call, we supply a new linear variable.

### 3.3.1. Regular typing rules

Before we get to the typing rules, we must first create a few predicates on contexts, and then extend the notion of context disjointness.

**Definition 3.3.1.** We define two predicates on types and contexts. Let copy$(\tau)$ if $\tau$ is a copy type, and let linear$(\tau)$ if $\tau$ is a linear type. Let copy$(\Gamma)$ if $\Gamma$ contains only copy bindings, i.e. $\Gamma(x) = \tau \implies \text{copy}(\tau)$ for all $x$. Let linear$(\Gamma)$ if $\Gamma$ contains only linear bindings, i.e. $\Gamma(x) = \tau \implies \text{linear}(\tau)$ for all $x$.

**Definition 3.3.2.** Let $\Gamma(\mathbb{1})$ be the linear part of the context $\Gamma$, and $\Gamma(\mathbb{0})$ be the copy part of the context. In other words, they are contexts such that linear$(\Gamma(\mathbb{1}))$, copy$(\Gamma(\mathbb{0}))$ and $\Gamma = \Gamma(\mathbb{1}) \cup \Gamma(\mathbb{0})$.

**Definition 3.3.3.** We extend the notion of context disjointness to work with copy types as well as linear types. We define $\Gamma_1 \perp \Gamma_2$ whenever $\Gamma_1(\mathbb{1}) \perp \Gamma_2(\mathbb{1})$, using the previous notion of disjointness for linear languages. In other words, disjointness is not affected by copy variables; they may appear in both contexts.

Using these definitions, we can define the typing rules, which can be found below. They mix the behaviours of both previous languages, while taking care of the interactions described earlier.

- The $\text{var}_{\text{b}}^{\text{reg}}$ rule requires the corresponding variable to be either copy, or the only linear variable in the context.

- The func $\text{copy}_{\text{b}}^{\text{reg}}$ rule handles the copy functions. It requires that no linear variables (other than the argument variable) exist in the context. The rest is like the function rule from $\lambda$.
- The func $\text{linear}_{\text{b}}^{\text{reg}}$ rule handles the linear functions. It has no further restrictions, since the function itself is linear already.
- The $\text{app}_{\text{b}}^{\text{reg}}$ rule handles function application. Because of how we defined context disjointness in mixed languages, this rule now nicely combines both behaviours.

$$\frac{\text{copy}(\Gamma \setminus \{(x,\tau)\})}{\Gamma \cup \{(x,\tau)\} \vdash_{\text{b}} x : \tau}\ (\text{var}_{\text{b}}^{\text{reg}})$$

$$\frac{\text{copy}(\Gamma) \qquad \Gamma \cup \{(x,\tau_1)\} \vdash_{\text{b}} e : \tau_2}{\Gamma \vdash_{\text{b}} \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2}\ (\text{func copy}_{\text{b}}^{\text{reg}})$$

$$\frac{\Gamma \cup \{(x,\tau_1)\} \vdash_{\text{b}} e : \tau_2}{\Gamma \vdash_{\text{b}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2}\ (\text{func linear}_{\text{b}}^{\text{reg}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\text{b}} e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash_{\text{b}} e_2 : \tau_1}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{b}} e_1\ e_2 : \tau_2}\ (\text{app}_{\text{b}}^{\text{reg}})$$

**Exercise 3.3.4.** Write down proof trees for the examples that are well-typed. Decide yourself which context is needed to get a correct derivation.

We can now formalise a very interesting property about the lambda calculi we defined, namely that $\lambda^{\mathbb{1}}$ is the "smallest" language, and $\lambda$ is the "biggest" language.

**Definition 3.3.5.** Let $e$ be an expression in $\lambda^{\mathsf{b}}$, and $\tau$ be a type in $\lambda^{\mathsf{b}}$. Then $e^{(\mathbb{1})}$ and $\tau^{(\mathbb{1})}$ are the corresponding expression and type in $\lambda^{\mathbb{1}}$ where all copy constructions have been replaced by equivalent linear constructions. For example, a copy function $\lambda x_\tau.\ e$ is transformed into $\lambda^{\mathbb{1}} x_{\tau^{(\mathbb{1})}}.\ e^{(\mathbb{1})}$. Likewise, $e^{(\mathbb{0})}$ and $\tau^{(\mathbb{0})}$ are the corresponding expression and type in $\lambda$ where all linear constructions have been replaced by equivalent copy constructions. Equivalent operations on contexts – $\Gamma^{(\mathbb{1})}$ and $\Gamma^{(\mathbb{0})}$ – also exist.

**Theorem 3.3.6** ($\lambda^{\mathbb{1}} \le \lambda^{\mathsf{b}} \le \lambda$)**.** For $e$, $\tau$ and $\Gamma$ in $\lambda^{\mathsf{b}}$, we have

$$\Gamma^{(\mathbb{1})} \vdash_{\mathbb{1}} e^{(\mathbb{1})} : \tau^{(\mathbb{1})} \implies \Gamma \vdash_{\text{b}} e : \tau$$

and

$$\Gamma \vdash_{\text{b}} e : \tau \implies \Gamma^{(\mathbb{0})} \vdash_{\lambda} e^{(\mathbb{0})} : \tau^{(\mathbb{0})}.$$

*Proof.* This relation follows from the fact that the typing rules of $\lambda^{\mathsf{b}}$ are designed to be more permissive than those of $\lambda^{\mathbb{1}}$. In other words, all linear derivations in $\lambda^{\mathbb{1}}$ may also be made in $\lambda^{\mathsf{b}}$. Likewise, the rules of $\lambda$ are more permissive than those of $\lambda^{\mathsf{b}}$, by design. This is because linear derivations are just copy derivations with more restrictions, and transforming to $\lambda$ removes those restrictions. □

### 3.3.2. Algorithmic typing rules

Before we can define the algorithmic typing rules, we make an operator. This operator helps us decide whether a function argument has been used correctly; if a function argument is linear, it must not be in the resulting context anymore. If the function argument is copy, there is no restriction.

**Definition 3.3.7.** Let $\text{used}(x, \Gamma)$ if a variable has been "used up" correctly. That is, if either $\Gamma(x) =$ Nothing, or if $\Gamma(x) = \tau$ such that $\text{copy}(\tau)$.

We now define the algorithmic typing rules. For clarity, the rule for variables is split into two.

$$\frac{\Gamma(x) = \tau \qquad \text{copy}(\tau)}{\Gamma \vdash_{\text{b}} x : \tau \dashv \Gamma}\ (\text{var copy}_{\text{b}}^{\text{alg}})$$

$$\frac{\Gamma(x) = \tau \qquad \text{linear}(\tau)}{\Gamma \vdash_{\mathrm{b}} x : \tau \dashv \Gamma - x}\ (\text{var linear}^{\text{alg}}_{\text{b}})$$

$$\frac{\Gamma_1(\mathbb{0}) \cup (x, \tau_1) \vdash_{\mathrm{b}} e : \tau_2 \dashv \Gamma_2 \qquad \text{used}(x, \Gamma_2)}{\Gamma_1 \vdash_{\mathrm{b}} \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2 \dashv \Gamma_1}\ (\text{func copy}^{\text{alg}}_{\text{b}})$$

$$\frac{\Gamma_1 \cup \{(x, \tau_1)\} \vdash_{\mathrm{b}} e : \tau_2 \dashv \Gamma_2 \qquad \text{used}(x, \Gamma_2)}{\Gamma_1 \vdash_{\mathrm{b}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2 \dashv \Gamma_2 - x}\ (\text{func linear}^{\text{alg}}_{\text{b}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{b}} e_1 : \tau \dashv \Gamma_2 \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \qquad \Gamma_2 \vdash_{\mathrm{b}} e_2 : \tau_1 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{b}} e_1\ e_2 : \tau_2 \dashv \Gamma_3}\ (\text{app}^{\text{alg}}_{\text{b}})$$

There is one important note in the func $\text{copy}^{\text{alg}}_{\text{b}}$ rule. The rule ends with the resulting context $\Gamma_1$. This is because a copy expression does not remove (free) variables from the context. Therefore, $\Gamma_2$ is just $\Gamma_1(\mathbb{0}) \cup (x, \tau_1)$, so having $\Gamma_1$ as resulting context is not a mistake.

**Exercise 3.3.8.** Write down proof trees for the examples that are well-typed, this time using the algorithmic typing rules. Decide yourself which initial context is needed to get a correct derivation, and show what resulting context this yields.

Like in the previous section, the regular and algorithmic typing rules have the same notion of well-typing. We formalise this with a theorem, but will leave a proof to the reader, since it goes by the same line as the proof in Theorem 3.2.8.

**Theorem 3.3.9.** The regular and algorithmic typing rules for $\lambda^{\mathsf{b}}$ have the same notion of well-typing. More formally,

$$\Gamma \vdash_{\mathrm{b}} e : \tau \iff \Gamma \vdash_{\mathrm{b}} e : \tau \dashv \Gamma(\mathbb{0}).$$

Now we can also state a useful corollary.

**Corollary 3.3.10** ($\lambda^{\mathbb{1}} \leq \lambda^{\mathsf{b}} \leq \lambda$ for algorithmic typing)**.** For $e$, $\tau$ and $\Gamma$ in $\lambda^{\mathsf{b}}$, we have

$$\Gamma^{(\mathbb{1})} \vdash_{\mathbb{1}} e^{(\mathbb{1})} : \tau^{(\mathbb{1})} \dashv \emptyset \implies \Gamma \vdash_{\mathrm{b}} e : \tau \dashv \Gamma(\mathbb{0})$$

and

$$\Gamma \vdash_{\mathrm{b}} e : \tau \dashv \Gamma(\mathbb{0}) \implies \Gamma^{(\mathbb{0})} \vdash_{\lambda} e^{(\mathbb{0})} : \tau^{(\mathbb{0})}.$$

*Proof.* A direct consequence of the previous theorems. □

## 3.3.3. Type checker

We end the section with the algorithmic type checking function for $\lambda^{\mathsf{b}}$, and some theorems related to it. Since proofs in this subsection are analogous to proofs seen earlier, we will not give any proofs for the following theorems. We encourage you to look at the theorems, however, and see if you agree with them.

The type checker can be found below:

$$\text{check}_{\mathrm{b}}\,(x, \Gamma) = \begin{cases} (\tau, \Gamma) & \text{if } \Gamma(x) = \tau \text{ and copy}(\tau); \\ (\tau, \Gamma \setminus (x, \tau)) & \text{otherwise, if } \Gamma(x) = \tau; \\ \text{Error} & \text{otherwise.} \end{cases}$$

$$\text{check}_{\mathrm{b}}\,(\lambda x_{\tau_1}.\ e, \Gamma_1) = \begin{cases} (\tau_1 \to \tau_2, \Gamma_1) & \text{if check}_{\mathrm{b}}\,(e, \Gamma_1(\mathbb{0}) \cup \{(x, \tau_1)\}) = (\tau_2, \Gamma_2) \text{ and used}(x, \Gamma_2); \\ \text{Error} & \text{otherwise.} \end{cases}$$

$$\text{check}_{\mathrm{b}}\,(\lambda^{\mathbb{1}} x_{\tau_1}.\ e, \Gamma_1) = \begin{cases} (\tau_1 \multimap \tau_2, \Gamma_2) & \text{if check}_{\mathrm{b}}\,(e, \Gamma_1 \cup \{(x, \tau_1)\}) = (\tau_2, \Gamma_2) \text{ and used}(x, \Gamma_2); \\ \text{Error} & \text{otherwise.} \end{cases}$$

$$\text{check}_{\mathrm{b}}\,(e_1\ e_2, \Gamma_1) = \begin{cases} (\tau_2, \Gamma_3) & \begin{array}{l}\text{if check}_{\mathrm{b}}\,(e_1, \Gamma_1) = (\tau, \Gamma_2),\ \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \\ \qquad\qquad \text{and check}_{\mathrm{b}}\,(e_2, \Gamma_2) = (\tau_1, \Gamma_3);\end{array} \\ \text{Error} & \text{otherwise.} \end{cases}$$

Now we give the usual equivalence theorem and the “subset” corollary for type checking:

**Theorem 3.3.11** (check implements the rules of $\lambda^{\mathsf{b}}$)**.** We have the equivalence

$$\mathrm{check}_{\mathrm{b}}\,(e, \Gamma_1) = (\tau, \Gamma_2) \iff \Gamma_1 \vdash_{\mathrm{b}} e : \tau \dashv \Gamma_2.$$

**Corollary 3.3.12** ($\lambda^{\mathbb{1}} \leq \lambda^{\mathsf{b}} \leq \lambda$ for type checking functions)**.** For $e$, $\tau$ and $\Gamma$ in $\lambda^{\mathsf{b}}$, we have

$$\mathrm{check}_{\mathbb{1}}\,(e^{(\mathbb{1})}, \Gamma^{(\mathbb{1})}) = (\tau^{(\mathbb{1})}, \emptyset) \implies \mathrm{check}_{\mathrm{b}}\,(e, \Gamma) = (\tau, \Gamma(\mathbb{0}))$$

and

$$\mathrm{check}_{\mathrm{b}}\,(e, \Gamma) = (\tau, \Gamma(\mathbb{0})) \implies \mathrm{check}_{\lambda}\,(e^{(\mathbb{0})}, \Gamma^{(\mathbb{0})}) = \tau^{(\mathbb{0})}.$$

## 3.4. Shadowing

In the last sections, we have assumed Barendregt's Variable Convention [2, 20, 21]. However, when writing programs, this is an impractical convention. We would instead like to shadow variables, without affecting the semantics of the typing rules. Shadowing a variable $x$ into type $\tau$ consists of the following steps, assuming we start with context $\Gamma_1$:

(a) take note of the previous type of $x$, if it exists;

(b) let $\Gamma_1(x) := \tau$;

(c) run the scope in which $x$ is shadowed, yielding $\Gamma_2$;

(d) if $\tau$ is linear, ensure that $\Gamma_2(x) = \text{Nothing}$;

(e) restore the previous state of $x$, as remembered in step (a).

Step (d) is integral; it ensures that step (e) does not accidentally delete a linear variable from the context. Below we provide some examples for when an expression should be considered well-typed, assuming we may shadow now.

| **Expression** | **Well-typed** |
|---|---|
| $\lambda x_{\alpha_0}.\ x$ | yes |
| $\lambda x_{\alpha_0}.\ (\lambda x_{\alpha_0}.\ x)$ | yes |
| $\lambda x_{\alpha_0}.\ (\lambda x_{\alpha_1}.\ x)$ | yes |
| $\lambda x_{\alpha_0}.\ (\lambda x_{\alpha_0^{\mathbb{1}}}.\ x)$ | yes |
| $\lambda x_{\alpha_0^{\mathbb{1}}}.\ (\lambda x_{\alpha_0}.\ x)$ | no: linear variable unused |
| $\lambda x_{\alpha_0^{\mathbb{1}}}.\ (\lambda x_{\alpha_0}.\ x)\ x$ | no: argument type mismatch |
| $\lambda x_{\alpha_0^{\mathbb{1}}}.\ (\lambda x_{\alpha_0^{\mathbb{1}}}.\ x)\ x$ | yes |

Figure 3.2: Examples of $\lambda^{\mathsf{b}}$-expressions where shadowing is allowed, evaluated with initial context $\Gamma$.

Let us now create notation that formalises these steps:

**Definition 3.4.1.** Let $\Gamma \leftarrow (x, \tau)$ *set a variable* $x$ to type $\tau$ in the context $\Gamma$. Formally, the operation $\Gamma \leftarrow (x, \tau)$ yields the context $(\Gamma - x) \cup \{(x, \tau)\}$. We may also set a variable to Nothing instead of $\tau$, in which case the operation yields the context $\Gamma - x$.

**Definition 3.4.2.** Given an algorithmic typing judgement $\Gamma_1 \vdash e : \tau \dashv \Gamma_2$, let $\Gamma_1 \mid x : \tau_x \vdash e : \tau \dashv \Gamma_2$ be an equivalent judgement where $x$ is shadowed with type $\tau_x$, i.e.

$$\begin{aligned}\Gamma_1 \mid x : \tau_x \vdash e : \tau \dashv \Gamma_2 \iff & \exists_{\Gamma_3}[\Gamma_1 \leftarrow (x, \tau_x) \vdash e : \tau \dashv \Gamma_3 \\ & \wedge \text{used}(x, \Gamma_3) \\ & \wedge \Gamma_2 = \Gamma_3 \leftarrow (x, \Gamma_1(x))].\end{aligned}$$

Since shadowing is only really an issue in type checking implementations, we do not create a shadowing judgement for regular typing rules.

We can also generalise this shadowing notation to an entire context, instead of just one variable binding:

**Definition 3.4.3.** Given an algorithmic typing judgement $\Gamma_1 \vdash e : \tau \dashv \Gamma_2$, let $\Gamma_1 \mid \Gamma \vdash e : \tau \dashv \Gamma_2$ be an equivalent judgement where all variables in $\Gamma$ are shadowed with their respective types. Formally, if $(x_1, \tau_1), \ldots, (x_n, \tau_n)$ are all bindings in $\Gamma$, we get

$$\begin{aligned}\Gamma_1 \mid \Gamma \vdash e : \tau \dashv \Gamma_2 \iff & \exists_{\Gamma_3}[\Gamma_1 \leftarrow (x_1, \tau_1) \leftarrow \ldots \leftarrow (x_n, \tau_n) \vdash e : \tau \dashv \Gamma_3 \\ & \wedge \forall_{1 \leq i \leq n}[\text{used}(x_i, \Gamma_3)] \\ & \wedge \Gamma_2 = \Gamma_3 \leftarrow (x, \Gamma_1(x_1)) \leftarrow \ldots \leftarrow (x_n, \Gamma_1(x_n))].\end{aligned}$$

*Remark.* Instead of writing the context $\Gamma$ in the above definition, we may also write out a list of bindings. This gives us the notation $\Gamma_1 \mid x_1 : \tau_1 \mid \cdots \mid x_n : \tau_n \vdash e : \tau \dashv \Gamma_2$.

### 3.4.1. Using shadowing

We now modify the function rules of $\lambda^{\mathsf{b}}$ — since they introduce variables — to support shadowing, and show that this change does not affect the semantics of the language.

$$\frac{\Gamma_1(\mathbb{0}) \mid x : \tau_1 \vdash_{\mathrm{b}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{b}} \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2 \dashv \Gamma_1} \text{(func copy shadow}_{\mathrm{b}}^{\mathrm{alg}}\text{)}$$

$$\frac{\Gamma_1 \mid x : \tau_1 \vdash_{\mathrm{b}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{b}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2 \dashv \Gamma_2} \text{(func linear shadow}_{\mathrm{b}}^{\mathrm{alg}}\text{)}$$

As you may see, the rules are more concise than the original rules func copy$_{\mathrm{b}}^{\mathrm{alg}}$ and func linear$_{\mathrm{b}}^{\mathrm{alg}}$. This is because the shadowing operator takes care of the "used" constraint and variable restoration (for the linear rule).

The following theorem relates the typing rules of $\lambda^{\mathsf{b}}$ with shadowing to those without shadowing.

**Theorem 3.4.4** (shadowing acts like $\alpha$-renaming)**.** Let $e_x$ be an expression in which at some point $x$ is shadowed. Let $e$ be the same expression as $e_x$, but where the shadowed parts of $x$ have been replaced by (formally known as *$\alpha$-renamed to*) a new variable $y$, such that no shadowing takes place anymore. Then, we get the equivalence $\Gamma_1 \vdash_{\mathrm{b}} e_x : \tau \dashv \Gamma_2 \iff \Gamma_1 \vdash_{\mathrm{b}} e : \tau \dashv \Gamma_2$, where the former judgement uses the shadowing rules, and the latter judgement uses the original rules.

**Exercise 3.4.5.** Try to prove this theorem using induction on the construction of the expression, like in previous proofs.

## 3.5. Bidirectional type checking

At the moment, we have required type annotations for function arguments. However, in some situations the argument type can be inferred. We would like to create new typing rules, so that argument types can be inferred.

In more complex languages, there could be more places where type annotations are required, yet can be inferred. The strategy we present in this section is independent of the place in the language where the problem lies, so it can provide inference for more than just function argument types.

Let us start off by extending the syntax of $\lambda^\flat$ with optional type argument type annotations $x : \tau$ and expression type annotations $e : \tau$. Only one function type is needed – the copy function – since annotations will allow us to "downgrade" a copy function to a linear function if needed. This is formally called "subsumption", and will be explained later.

$$i, j \in \mathbb{N}, \quad x, y \in \text{Var}$$

$$\begin{aligned}
\tau &::= \alpha_i \mid \alpha_i^{\mathbb{1}} \mid \tau \to \tau \mid \tau \multimap \tau \\
\chi &::= x \mid x : \tau \\
e &::= e : \tau \mid x \mid \lambda\chi.\ e \mid e\ e
\end{aligned}$$

Below we provide some examples for this new syntax:

| Expression | Well-typed | Inference |
|---|---|---|
| $\lambda x : \alpha_0.\ x$ | yes | - |
| $\lambda x.\ x$ | no: cannot infer type of $x$ | - |
| $(\lambda y.\ y)\ x$ | yes | $y \mapsto \Gamma(x)$ |
| $x : \tau$ | if $\Gamma(x) = \tau$ | - |
| $(\lambda y.\ y)\ x$ | yes | $y \mapsto \Gamma(x)$ |
| $((\lambda y.\ y)\ x) : \tau$ | if $\Gamma(x) = \tau$ | $y \mapsto \Gamma(x)$ |
| $(\lambda x.\ x) : \tau \to \tau$ | yes | $x \mapsto \tau$ |
| $(\lambda x.\ x) : \tau \multimap \tau$ | yes: copy function can be used exactly once | $x \mapsto \tau$ |
| $((\lambda x.\ x) : \tau \multimap \tau) : \tau \to \tau$ | no: linear function may not become copy | - |
| $(\lambda x : \tau \multimap \tau.\ x)\ (x : \tau \to \tau)$ | yes: copy argument can be used exactly once | - |
| $(\lambda x : \tau \to \tau.\ x)\ (x : \tau \multimap \tau)$ | no: linear argument may not become copy | - |

The fourth to last and second to last examples show subsumption in action. It is integral to making the type checking work for this syntax. Without it, any benign type mismatch during type checking could mess up the process. Subsumption allows us to talk in terms of properties, instead of concrete types: if we need to type check a linear function, we don't need a concrete linear type, but just a type that can work with the linearity requirements.

### 3.5.1. The recipe

In Bidirectional Typing [3], the problem of unnecessary type annotations is solved by separating inference/calculation from checking/verifying. There, typing rules specify if the rule is trying to infer the type of the expression, or whether they're testing an expression against an already known type. This allows the typing rules to go for the easier checking step, whenever there is enough (redundant) information to know the type beforehand, which could remove the need for some annotations.

We introduce these two separated concepts as typing judgements. The notation is the same as in Bidirectional Typing, but has been adapted for algorithmic typing.

**Definition 3.5.1.** Let $\Gamma_1 \vdash e \Rightarrow \tau \dashv \Gamma_2$ be an *inferring typing judgement*, and $\Gamma_1 \vdash e \Leftarrow \tau \dashv \Gamma_2$ be a *checking typing judgement*.

**Definition 3.5.2.** Let $\tau_1 \;<:\; \tau_2$ ($\tau_1$ *subsumes* $\tau_2$) if $\tau_1$ can "act like" $\tau_2$. This operator is reflexive and transitive.

*Remark.* For $\lambda^{\mathsf{b}}$, exactly the following subsumptions hold:

- $\tau \;<:\; \tau$, because of reflexivity;
- $\tau_2 \to \tau_3 \;<:\; \tau_1 \multimap \tau_4$ whenever $\tau_1 \;<:\; \tau_2$ and $\tau_3 \;<:\; \tau_4$. Namely, for an expression of type $\tau_2 \to \tau_3$ to act like $\tau_1 \multimap \tau_4$, we must allow arguments of type $\tau_1$, which can act like $\tau_2$. This yields us type $\tau_3$, which can then act like $\tau_4$.

**Exercise 3.5.3.** Convince yourself that the subsumption relation is indeed transitive for $\lambda^{\mathsf{b}}$.

The two basic typing rules are about switching between checking and inference. Type annotations let expressions be type checked using the more permissive checking rule. Subsumption allows a checking rule to use infer-only rules (and verify that the inferred type is compatible). These basic rules can be formulated for a lot of languages, but below, we provide them for $\lambda^{\mathsf{b}}$:

$$\frac{\Gamma_1 \vdash_{\mathrm{b}} e \Leftarrow \tau \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{b}} e : \tau \Rightarrow \tau \dashv \Gamma_2}\ (\mathrm{anno}^{\mathrm{bi}}_{\mathrm{b}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{b}} e \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \tau_1 \;<:\; \tau_2}{\Gamma_1 \vdash_{\mathrm{b}} e \Leftarrow \tau_2 \dashv \Gamma_2}\ (\mathrm{subsumes}^{\mathrm{bi}}_{\mathrm{b}})$$

To transform algorithmic typing rules into bidirectional typing rules, we use the "recipe" described in Bidirectional Typing [3]. This recipe can be summarised as follows:

(a) Inference judgements may be seen as partial functions taking an initial context and an expression, yielding a type and a resulting context;

(b) Checking judgements may be seen as partial functions taking an initial context, expression and type, yielding a resulting context;

(c) Suppose $\Gamma_1 \vdash e : \tau \dashv \Gamma_2$ is at the conclusion of the algorithmic typing rule we wish to transform. As stated above, we interpret this conclusion as a function;

(d) This conclusion function computes the resulting context (and possibly the type) using the premise of the typing rule. For this, the typing judgements in the premise are computed in a certain order;

(e) Each typing judgement in the premise of the typing rule may be transformed into either an inference or a checking rule. If the type of such judgement can be computed from all current inputs (including its initial context and expression), the judgement becomes a checking rule. Otherwise, the judgement becomes an inference rule, giving the type as input for any next judgements. The resulting context also becomes an input for any next judgements;

(f) If the conclusion function can be computed without the type $\tau$, this computation becomes an inference rule;

(g) If adding $\tau$ as an input gives rise to a sufficiently "alternative" computation, this new computation becomes a checking rule.

If we look at the two basic rules from a bit earlier, these conform with the recipe: the annotation rule $\mathrm{anno}^{\mathrm{bi}}_{\mathrm{b}}$ has $\tau$ as input, since it's part of the input expression $e : \tau$. Therefore, the premise can use a checking rule, and we can always infer for the conclusion. A checking version would make no sense, since we have the subsumption rule.

The subsumption rule $\mathrm{subsumes}^{\mathrm{bi}}_{\mathrm{b}}$ cannot get an inference version, since we do not know what type $\tau_2$ to subsume to. The existing checking version makes sense, since $\tau_1$ is not known from the input, so must be inferred.

The reason why step (g) requires the computation to be sufficiently 'alternative', is because for many checking rules, if they fail, we can just try the corresponding inference rule (if it exists) via the subsumption rule; so even without an explicit checking rule, checking is always at least as strong as inferring.

Also, in most cases you would expect that the checking rule can type check strictly more cases than the inference rule, since we have more information available. So in a type cheker, failure during a checking rule often means you can abort, so that checking the inference rule via subsumption is not necessary. In some cases, however, the "alternative" checking computation is not strictly stronger than the inference computation. There are not a lot of these cases, and throughout this thesis, we will explicitly mention cases that do not adhere to this principle.

### 3.5.2. Bidirectionalising $\lambda^{\mathsf{b}}$

If the recipe sounds a bit abstract, do not worry, for in this subsection, we will give a concrete example. Namely, we will be creating bidirectional typing rules for $\lambda^{\mathsf{b}}$. We start with the simplest rules, those for variables.

$$\frac{\Gamma(x) = \tau \qquad \text{copy}(\tau)}{\Gamma \vdash_{\text{b}} x \Rightarrow \tau \dashv \Gamma} \text{(var copy}_{\text{b}}^{\text{bi}}\text{)}$$

$$\frac{\Gamma(x) = \tau \qquad \text{linear}(\tau)}{\Gamma \vdash_{\text{b}} x \Rightarrow \tau \dashv \Gamma - x} \text{(var linear}_{\text{b}}^{\text{bi}}\text{)}$$

These rules are the same as var $\text{copy}_{\text{b}}^{\text{alg}}$ and var $\text{linear}_{\text{b}}^{\text{alg}}$, but written as inference rules instead of algorithmic rules. This is because the type to infer is stored in the initial context $\Gamma$. There are no more efficient cases where the rule could work without $\tau$ being inside of this context, so a checking rule is skipped.

Now as for the function rules func $\text{copy}_{\text{b}}^{\text{alg}}$ and func $\text{linear}_{\text{b}}^{\text{alg}}$, they do get both inference and checking rules. We start with the inference rules:

$$\frac{\Gamma_1(\mathbb{0}) \mid x : \tau_1 \vdash_{\text{b}} e \Rightarrow \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{b}} \lambda x : \tau_1 .\ e \Rightarrow \tau_1 \to \tau_2 \dashv \Gamma_1} \text{(func copy}{\Rightarrow}_{\text{b}}^{\text{bi}}\text{)}$$

$$\frac{\Gamma_1 \mid x : \tau_1 \vdash_{\text{b}} e \Rightarrow \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{b}} \lambda^{\mathbb{1}} x : \tau_1 .\ e \Rightarrow \tau_1 \multimap \tau_2 \dashv \Gamma_2} \text{(func linear}{\Rightarrow}_{\text{b}}^{\text{bi}}\text{)}$$

In this case, we need argument type annotations to figure out $\tau_1$, and must use an infer rule to compute $\tau_2$. Now for the checking rules:

$$\frac{\Gamma_1(\mathbb{0}) \mid x : \tau_1 \vdash_{\text{b}} e \Leftarrow \tau_2 \dashv \Gamma_2 \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\}}{\Gamma_1 \vdash_{\text{b}} \lambda x .\ e \Leftarrow \tau \dashv \Gamma_1} \text{(func copy}{\Leftarrow}_{\text{b}}^{\text{bi}}\text{)}$$

$$\frac{\Gamma_1 \mid x : \tau_1 \vdash_{\text{b}} e \Leftarrow \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{b}} \lambda^{\mathbb{1}} x .\ e \Leftarrow \tau_1 \multimap \tau_2 \dashv \Gamma_2} \text{(func linear}{\Leftarrow}_{\text{b}}^{\text{bi}}\text{)}$$

Since both $\tau_1$ and $\tau_2$ are known now from the input type, we can omit both the argument type annotation and use a checking rule in the premises. Both changes are significant — the former because a change in syntax is significant, and the latter because checking rules are more permissive because of subsumption – which is why we defined these checking versions.

In the rule func $\text{copy}{\Leftarrow}_{\text{b}}^{\text{bi}}$, $\tau$ may be both bopy or linear. This is much like the subsumption rule, but we need to state this explicitly since subsumption doesn't go from checking rule to checking rule.

Lastly, we get to function application:

$$\frac{\Gamma_1 \vdash_{\text{b}} e_1 \Rightarrow \tau \dashv \Gamma_2 \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \qquad \Gamma_2 \vdash_{\text{b}} e_2 \Leftarrow \tau_1 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{b}} e_1\ e_2 \Rightarrow \tau_2 \dashv \Gamma_3} \text{(app}{\Rightarrow}_{\text{b}}^{\text{bi}}\text{)}$$

$$\frac{\Gamma_1 \vdash_{\text{b}} e_2 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{b}} e_1 \Leftarrow \tau_1 \multimap \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{b}} e_1\ e_2 \Leftarrow \tau_2 \dashv \Gamma_3} \text{(app}{\Leftarrow}_{\text{b}}^{\text{bi}}\text{)}$$

The inference rule requires the function expression $e_1$ to be inferred. If $e_2$ were to be inferred instead, we would not get info about $\tau_2$. In the checking rule, we do know $\tau_2$ now, so we can switch which subexpression needs to be inferred now.

Note that a type checker should be careful with this checking rule, since this is a rule that breaks the principle of checking being strictly stronger than inference. Namely, in most cases a failing checking rule implies that the inference rule will also fail. However, this is not the case with function application, as defined here. If the checking rule fails because $e_2$ can only be checked, we might still be able to type check using the inference rule and subsumption rule.

**Exercise 3.5.4.** In the checking rules for function, we disallow argument type annotations. We could add rules, or change the existing rules, so that argument type annotations are still allowed (and thus serve as a 'sanity check' in the checking rule). What would these rules look like?

To link the bidirectional and algorithmic rules together, we have two theorems:

**Theorem 3.5.5** (soundness)**.** Suppose $e_1$ is an expression with argument type annotations in all possible places, and $e_2$ is the same expression, without any other type annotations. Then we get that $\Gamma_1 \vdash_{\mathrm{b}} e_1 \Leftarrow \tau \dashv \Gamma_2$ implies $\Gamma_1 \vdash_{\mathrm{b}} e_2 : \tau \dashv \Gamma_2$.

**Theorem 3.5.6** (completeness)**.** Suppose $e$ is an expression with argument type annotations in all possible places and no other type annotations. Then $\Gamma_1 \vdash_{\mathrm{b}} e : \tau \dashv \Gamma_2$ implies $\Gamma_1 \vdash_{\mathrm{b}} e \Rightarrow \tau \dashv \Gamma_2$.

Chapter 4.

# Algorithmic MPGV typing rules

In this chapter, we build algorithmic, and then bidirectional typing rules for the MPGV language. For this, we base ourselves on the definitions in Multiparty GV [12]. These bidirectional rules form the basis of the interpreter, which is described in chapter 5.

In section 4.1, we recall the syntax of MPGV, and describe changes made. In section 4.2, we use the regular typing rules and the strategies from the background (chapter 3) to derive algorithmic typing rules. Some intricacies in and changes to the regular typing rules will also be touched upon. Lastly, in section 4.3, we use the algorithmic typing rules and the bidirecionalisation recipe from section 3.5 to derive bidirectional typing rules.

## 4.1. Recalling the syntax

Before we can talk about typing rules, we must know the formal syntax of the MPGV language. Below, we present the syntax and variable conventions for MPGV as used throughout this thesis. To get familiar with the syntax, we encourage to compare it with the code examples in Chapter 2; the syntax used in the interpreter is virtually identical to the formal mathematical syntax.

$$i, j \in \mathbb{N}, \quad x, y, f \in \text{Var}, \quad l, l' \in \text{Label}, \quad I \subseteq \text{Label}, \quad p, q \in \text{Part}, \quad \pi : \text{Part} \rightharpoonup \text{Part}$$

$$\begin{aligned}
\tau \in \text{Type} &::= \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \{l:\ \tau\}_{l\in I} \mid \tau \times \tau \mid \tau \to \tau \mid \tau \multimap \tau \mid L \\
L \in \text{LType} &::= ![p]\tau.\ L \mid ?[p]\tau.\ L \mid ![p]\{l:\ \tau.\ L\}_{l\in I} \mid ?[p]\{l:\ \tau.\ L\}_{l\in I} \mid \text{End} \\
G \in \text{GType} &::= [p \to p]\tau.\ G \mid [p \to p]\{l:\ \tau.\ G\}_{l\in I} \mid \text{End}
\end{aligned}$$

$$\begin{aligned}
\odot \in \text{BinOp} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{==} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \\
e \in \text{Expr} &::= () \mid n \mid x \mid \textbf{not}\ e \mid e \odot e \mid (e, e) \mid \langle l : e\rangle_\tau \mid \lambda x_\tau.\ e \mid \lambda^{\mathbb{1}} x_\tau.\ e \mid \textbf{rec}\ f_\tau\ x_\tau.\ e \mid e\ e \mid \\
&\qquad \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{let}\ (x, x) = e\ \textbf{in}\ e \mid \textbf{match}\ e\ \textbf{with}\ \{l:\ x \mapsto e\}_{l\in I} \mid \\
&\qquad \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{fork}\langle G\rangle(x \mapsto e, \ldots) \mid \textbf{send}[p](e, e) \mid \textbf{send}[p](e, l, e) \mid \\
&\qquad \textbf{receive}[p](e) \mid \textbf{redirect}_L[\pi](e) \mid \textbf{close}(e)
\end{aligned}$$

Note that this syntax slightly differs from that used in Multiparty GV [12]:

- Many expressions have gotten required type annotations, so that the algorithmic typing rules can be computable. An example of this is $\langle l : e\rangle_\tau$, which originally was written as just $\langle l : e\rangle$. Notably, functions get a $\mathbb{1}$ annotation to indicate linearity, and the fork-operation requires a global type, from which local types for each channel are derived.
- The language has been formally expanded with binary operations and boolean-related constructions, like if-statements and a boolean type **B**. The examples used in Multiparty GV imply that they exist, so for completeness sake, we formally define them. These additions follow the same semantics as related syntactic elements in the Haskell language. Most notably, the logical operations `&&` and `||` are short-circuiting.
- The labeled type is written differently in this thesis to make it more consistent with local type notations, and to increase clarity, especially when written in type annotations. Multiparty GV has the notation $\sum_{l\in I}.\ \tau$, whereas we use the set notation $\{l:\ \tau\}_{l\in I}$.
- We allow both labeled and unlabeled sending/receiving of values. Unlabeled sending is not possible in Multiparty GV, but it would be a very practical feature. To make the interpreter more user-friendly, we formalised the concept of unlabeled sending/receiving, so that the interpreter does not need "syntactic sugar" for this.

Regular typing rules for most of the MPGV language were defined in Multiparty GV [12]. For a complete overview of the regular typing rules, however, we refer to the appendix, section A.1. In this appendix, we also slightly modified the existing typing rules, so they work with the syntactical changes described earlier. The biggest changes are made to the function rules func $\text{copy}^{\text{reg}}_{\text{M}}$ and func $\text{linear}^{\text{reg}}_{\text{M}}$ (where the rule for linear functions now requires the function to be linearly annotated), and to the fork rule $\text{fork}^{\text{reg}}_{\text{M}}$ (where the consistency-check is omitted since it's implied from the used global type). The rules from the appendix will be the main rules "reference", which we use to derive the algorithmic and bidirectional typing rules in this chapter.

## 4.2. Algorithmic typing rules

We assume familiarity with the content presented in the background, chapter 3. This way, we can focus more closely on MPGV-specific parts of the typing rules, without re-explaining typing rules shared by many lambda-calculi, and strategies like shadowing described earlier. Just like with the regular typing rules, a full overview of the algorithmic typing rules can be found in section A.2.

### 4.2.1. Basic lambda calculus

MPGV shares many elements of the mixed lambda calculus $\lambda^{\mathsf{b}}$ from section 3.3. Therefore, the var $\text{copy}_{\text{M}}^{\text{alg}}$, var $\text{linear}_{\text{M}}^{\text{alg}}$, func $\text{copy}_{\text{M}}^{\text{alg}}$, func $\text{linear}_{\text{M}}^{\text{alg}}$ and $\text{app}_{\text{M}}^{\text{alg}}$ rules should be self-explanatory.

The language also has some constants: the unit (), two booleans True and False and natural numbers $n$. These are not linear, and always have the same type — **1**, **B** and **N** respectively. Therefore, these translate much like the rule for copy variables. We present the translation below:

$$\frac{\text{copy}(\Gamma)}{\Gamma \vdash_{\text{M}} \text{constant} : \text{type}} \;(\text{const.}_{\text{M}}^{\text{reg}})$$

$$\frac{}{\Gamma \vdash_{\text{M}} \text{constant} : \text{type} \dashv \Gamma} \;(\text{const.}_{\text{M}}^{\text{alg}})$$

The rules used above are schemas, meaning that they define a range of similar typing rules. Which exact rules they define are mentioned near the schema's definition in the appendix.

The language also has pairs, and arithmetic operations on natural numbers. Their algorithmic rules — $\text{pair}_{\text{M}}^{\text{alg}}$ and $\text{arith.}_{\text{M}}^{\text{alg}}$ — are similar to the $\text{app}_{\text{M}}^{\text{alg}}$ rule, in that they follow the general strategy for translating binary operations. Recall the general strategy from Figure 3.1.

Recall that MPGV has recursive functions. The regular typing rule for recursive functions is:

$$\frac{\text{copy}(\Gamma) \qquad \Gamma \cup \{(f, \tau_1 \to \tau_2), (x, \tau_1)\} \vdash_{\text{M}} e : \tau_2}{\Gamma \vdash_{\text{M}} \textbf{rec}\ f_{\tau_1 \to \tau_2}\ x_{\tau_1}.\ e : \tau_1 \to \tau_2} \;(\text{rec}_{\text{M}}^{\text{reg}})$$

In this rule, we add two variables to the function body, one being the recursive function invocation, and the other being the function argument. In terms of typing rules, it is not that much different from the copy function rule func $\text{copy}_{\text{M}}^{\text{alg}}$, so the translation should be fairly evident:

$$\frac{\Gamma_1(\mathbb{0}) \mid f : \tau_1 \to \tau_2 \mid x : \tau_1 \vdash_{\text{M}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{M}} \textbf{rec}\ f_{\tau_1 \to \tau_2}\ x_{\tau_1}.\ e : \tau_1 \to \tau_2 \dashv \Gamma_1} \;(\text{rec}_{\text{M}}^{\text{alg}})$$

Lastly, MPGV has let-expressions, which create variables and break up pairs. Those rules — let $\text{var}_{\text{M}}^{\text{alg}}$ and let $\text{pair}_{\text{M}}^{\text{alg}}$ — are akin to the function rule func $\text{linear}_{\text{M}}^{\text{alg}}$.

### 4.2.2. Booleans

For boolean logic, we have the if-statement and various logical operations. We start with translating the if-statement:

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\text{M}} e_1 : \mathbf{B} \qquad \Gamma_2 \vdash_{\text{M}} e_2 : \tau \qquad \Gamma_2 \vdash_{\text{M}} e_3 : \tau}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{M}} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau} \;(\text{if}_{\text{M}}^{\text{reg}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 : \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_2 : \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash_{\text{M}} e_3 : \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau \dashv \Gamma_3} \;(\text{if}_{\text{M}}^{\text{alg}})$$

Note that the input and output contexts to both branches must be the same, to ensure that linear variables are used deterministically. In the regular rule, this was accomplished with just the input context being the same.

The other rules again follow the general strategy. These are the $\text{not}_{\text{M}}^{\text{alg}}$, $\text{comp.}_{\text{M}}^{\text{alg}}$, $\text{order}_{\text{M}}^{\text{alg}}$ and $\text{logic}_{\text{M}}^{\text{alg}}$ rules. Note that for the rule $\text{logic}_{\text{M}}^{\text{alg}}$ also uses the strategy from copy functions to ensure that the second subexpression to logical AND and logical OR is copy. The reason it must be copy, is because the operation may short-circuit and thus may sometimes skip evaluating the second expression. If that expression were linear, this destroys the guarantee that linear variables are used exactly once.

The comparison and ordering rules are limited to certain types. We define these using the following predicates:

**Definition 4.2.1.** A type $\tau$ is *equality comparable* ($\text{eqComp}(\tau)$) whenever it is of type $\mathbf{1}$, $\mathbf{N}$ or $\mathbf{B}$, or whenever it is a pair $\tau_1 \times \tau_2$ of equality comparable types $\tau_1$ and $\tau_2$, or whenever it is a labeled type $\{l : \tau_l\}_{l \in I}$ of equality comparable types $\tau_l$. Other types are not equality comparable, most notably function types.

**Definition 4.2.2.** A type $\tau$ is *orderable* ($\text{orderable}(\tau)$) whenever it is of type $\mathbf{N}$, or whenever it is a pair $\tau_1 \times \tau_2$ of orderable types $\tau_1$ and $\tau_2$. Other types are not orderable.

The reason functions are excluded is because checking for equality depends on one's view point. Should functions be syntactically equal, or syntactically equal apart from variable naming, or should they just be semantically equal? How would we check for semantic equivalence? By not having them be comparable, we eliminate this issue.

The orderable types are natural numbers and (nested) pairs of natural numbers, because we can apply a lexicographical ordering using the standard mathematical ordering on natural numbers.

### 4.2.3. Choice

As seen in section 2.4, MPGV supports labeled expressions and labeled types. Using the match-statement, we then get a system analogous to switch-statements in popular languages like C++. The two relevant typing rules are presented below:

$$\frac{\Gamma \vdash_{\mathrm{M}} e : \tau_l \qquad l \in I}{\Gamma \vdash_{\mathrm{M}} \langle l : e \rangle_{\{l' :\ \tau_{l'}\}_{l' \in I}} : \{l' :\ \tau_{l'}\}_{l' \in I}} \ (\text{choice}_{\mathrm{M}}^{\text{reg}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e : \{l :\ \tau_l\}_{l \in I} \qquad \forall_{l \in I}[\Gamma_2 \cup \{(x_l, \tau_l)\} \vdash_{\mathrm{M}} e_l : \tau]}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{match } e \textbf{ with } \{l :\ x_l \mapsto e_l\}_{l \in I} : \tau} \ (\text{match}_{\mathrm{M}}^{\text{reg}})$$

Note that the match-rule requires the set of labels to be non-empty. If it were empty, the expression would not be able to return a value when running it through the interpreter, and it would have to either error or wait indefinitely. We made this choice to ensure that the interpreter has the least amount of runtime errors possible (which are hard to debug), and catches most errors at the time of type checking instead (which are easier to debug). The rules in Multiparty GV [12] do support empty label sets, however.

The translation to algorithmic typing rules is as follows:

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \tau_l \dashv \Gamma_2 \qquad l \in I}{\Gamma_1 \vdash_{\mathrm{M}} \langle l : e \rangle_{\{l' :\ \tau_{l'}\}_{l' \in I}} : \{l' :\ \tau_{l'}\}_{l' \in I} \dashv \Gamma_2} \ (\text{choice}_{\mathrm{M}}^{\text{alg}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \{l :\ \tau_l\}_{l \in I} \dashv \Gamma_2 \qquad \forall_{l \in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l : \tau \dashv \Gamma_3]}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match } e \textbf{ with } \{l :\ x_l \mapsto e_l\}_{l \in I} : \tau \dashv \Gamma_3} \ (\text{match}_{\mathrm{M}}^{\text{alg}})$$

The choice rule translation is nothing special, apart from the type annotation. This is so that a type checker knows what other labels to add to the returned type. The match rule, however, is a bit more complex. Just as with if-statements, all branches must result in the same output context, given the same input context. Also, we have variable shadowing in each branch to take care of. The match-rule translation is a direct result of those if-statement and shadowing strategies.

### 4.2.4. Concurrency

Now we get to the novel part of MPGV: its concurrency support. We start off chronologically with the fork-rule, which is arguably the most complex one. The regular rule is as follows:

$$\frac{\Gamma_1 \perp \cdots \perp \Gamma_n \qquad \text{participants}(G) \subseteq \{0, \ldots, n\} \qquad \forall_{1 \le i \le n}[\Gamma_i \cup \{(x_i, G \downarrow i)\} \vdash_{\mathrm{M}} e_i : \mathbf{1}]}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash_{\mathrm{M}} \mathbf{fork}\langle G \rangle (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : G \downarrow 0} \; (\text{fork}_{\mathrm{M}}^{\text{reg}})$$

It differs from that in Multiparty GV. Namely, instead of having a consistency check between local types, we use the global type to derive a set of consistent local types $G \downarrow i$ using the projection operator. A detailed specification of the operator can be found in Multiparty GV [12], but for this chapter, it is enough to know that the projection of $G$ onto a participant $i$ returns a local type describing the global type from the participant's perspective.

Because we have a global type $G$, we have to verify that all participants in $G$ have corresponding expressions (participant 0 needs no expression, since its channel is the return value of the fork-expression). Each such expression $e_i$ should use the given channel argument (the one that has the projected local type) and then return unit.

The translation to algorithmic rules is now analogous to the function rule translations:

$$\frac{\text{participants}(G) \subseteq \{0, \ldots, n\} \qquad \forall_{1 \le i \le n}[\Gamma_i \mid x_i : G \downarrow i \vdash_{\mathrm{M}} e_i : \mathbf{1} \dashv \Gamma_{i+1}]}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{fork}\langle G \rangle (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : G \downarrow 0 \dashv \Gamma_{n+1}} \; (\text{fork}_{\mathrm{M}}^{\text{alg}})$$

The (un)labeled sending and receiving rules — send $\text{l.}_{\mathrm{M}}^{\text{alg}}$, send $\text{unl.}_{\mathrm{M}}^{\text{alg}}$, recv. $\text{l.}_{\mathrm{M}}^{\text{alg}}$ and recv. $\text{unl.}_{\mathrm{M}}^{\text{alg}}$ — look like a lot because of the large types, but it is very self-explanatory. Sending is akin to a binary operation like arithmetic, and receiving akin to a unary operation like a labeled expression.

The close-operation $\text{close}_{\mathrm{M}}^{\text{alg}}$ is also trivial. Note that the type of this operation is **1**, which is consistent with what the fork-operation wants as a return-value to the subexpresisons.

Lastly, the redirect-operation was not trivial to translate. Namely, look at the regular typing rule:

$$\frac{\Gamma \vdash_{\mathrm{M}} e : \pi(L)}{\Gamma \vdash_{\mathrm{M}} \mathbf{redirect}_L[\pi](e) : L} \; (\text{redirect}_{\mathrm{M}}^{\text{reg}})$$

In Multiparty GV [12], the rule does not have a local type annotation in the redirect-expression. For a type checker, type checking for a computed type $\pi(L)$ is not possible, so instead $e$ yields a type $L'$. Returning $\pi^{-1}(L')$, however, is not a good idea. Namely, the redirection need not be injective. Therefore, we need the returned local type as a type annotation, so that we can compute the type that $e$ should have. So now, the following rule can indeed be named "algorithmic", since a type checker would be able to compute the return type:

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \pi(L) \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{redirect}_L[\pi](e) : L \dashv \Gamma_2} \; (\text{redirect}_{\mathrm{M}}^{\text{alg}})$$

## 4.3. Bidirectional typing rules

The bidirectionalisation of the algorithmic typing rules will be based on the recipe as described in section 3.5. The goal of the bidirecionalisation is to reduce the amount of type annotations necessary, and to generalise type annotations for any expression. Therefore, we present some modifications to the MPGV syntax (changes are uncoloured):

$$\begin{aligned}
\chi \in \text{AnnoVar} &::= x \mid x : \tau \\
e \in \text{Expr} &::= e : \tau \mid () \mid n \mid x \mid \textbf{not } e \mid e \odot e \mid (e, e) \mid \langle l : e \rangle \mid \lambda \chi.\ e \mid \textbf{rec } \chi\ x.\ e \mid e\ e \mid \\
&\quad\ \textbf{let } x = e \textbf{ in } e \mid \textbf{let } (x, x) = e \textbf{ in } e \mid \textbf{match } e \textbf{ with } \{l :\ \chi \mapsto e\} \mid \\
&\quad\ \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \textbf{fork}\langle G \rangle (x \mapsto e, \ldots) \mid \textbf{send}[p](e, e) \mid \textbf{send}[p](e, l, e) \mid \\
&\quad\ \textbf{receive}[p](e) \mid \textbf{redirect}[\pi](e) \mid \textbf{close}(e)
\end{aligned}$$

The first change is the addition of a new variable $\chi$ — the *annotatable variable*. This variable indicates either a normal variable $x$, or an annotated variable $x : \tau$. The second change is the removal of most type annotations, adding general expression annotation syntax, and the use of annotatable variables in expressions.

We also redefine subsumption, to add support for pairs and labeled types. A labeled type acts like another, if they provide a subset of the functionality of the labeled type it acts like; they may not add functionality compared to the labeled type it acts like. Pairs subsume whenever its contents subsume pairwise. Function subsumption is like in the background section. Sending/receiving is like labeled type subsumption, in that a subset of the functionality must be provided, and nothing more. Lastly, types that are equal should also subsume eachother.

**Definition 4.3.1.** Subsumption for MPGV is formally defined as follows:

| **Description** | **Case** | **Condition** |
|---|---|---|
| labeled types | $\{l :\ \tau_l\}_{l \in I}\ <:\ \{l :\ \tau_l'\}_{l \in I'}$ | if $I \subseteq I'$ and $\forall_{l \in I}[\tau_l\ <:\ \tau_l']$ |
| pairs | $\tau_1 \times \tau_2\ <:\ \tau_1' \times \tau_2'$ | if $\tau_1\ <:\ \tau_1'$ and $\tau_2\ <:\ \tau_2'$ |
| linear functions | $\tau_1 \multimap \tau_2\ <:\ \tau_1' \multimap \tau_2'$ | if $\tau_1'\ <:\ \tau_1$ and $\tau_2\ <:\ \tau_2'$ |
| function mix | $\tau_1 \to \tau_2\ <:\ \tau_1' \multimap \tau_2'$ | if $\tau_1'\ <:\ \tau_1$ and $\tau_2\ <:\ \tau_2'$ |
| copy functions | $\tau_1 \to \tau_2\ <:\ \tau_1' \to \tau_2'$ | if $\tau_1'\ <:\ \tau_1$ and $\tau_2\ <:\ \tau_2'$ |
| labeled send | $![p]\{l :\ \tau_l.\ L_l\}_{l \in I}\ <:\ ![p]\{l :\ \tau_l'.\ L_l'\}_{l \in I'}$ | if $I \subseteq I'$, $\forall_{l \in I}[\tau_l\ <:\ \tau_l' \wedge L_l\ <:\ L_l']$ |
| unlabeled send | $![p]\tau.\ L\ <:\ ![p]\tau'.\ L'$ | if $\tau\ <:\ \tau'$ and $L\ <:\ L'$ |
| labeled receive | $?[p]\{l :\ \tau_l.\ L_l\}_{l \in I}\ <:\ ?[p]\{l :\ \tau_l'.\ L_l'\}_{l \in I'}$ | if $I \subseteq I'$, $\forall_{l \in I}[\tau_l\ <:\ \tau_l' \wedge L_l\ <:\ L_l']$ |
| unlabeled receive | $?[p]\tau.\ L\ <:\ ?[p]\tau'.\ L'$ | if $\tau\ <:\ \tau'$ and $L\ <:\ L'$ |
| general equality | $\tau\ <:\ \tau$ | always |

### 4.3.1. Basic lambda calculus

Annotation and subsumption rules anno$^{\text{bi}}_{\text{M}}$ and subsumes$^{\text{bi}}_{\text{M}}$ are provided for MPGV. These can be found in the overview in section A.3 The rules var copy$^{\text{bi}}_{\text{M}}$, var linear$^{\text{bi}}_{\text{M}}$, func$\Rightarrow^{\text{bi}}_{\text{M}}$, func$\Leftarrow^{\text{bi}}_{\text{M}}$, app$\Rightarrow^{\text{bi}}_{\text{M}}$ and app$\Leftarrow^{\text{bi}}_{\text{M}}$ are similar to the lambda calculus described in section 3.5.

However, the function rules now allow annotatable variables, instead of only unannotated variables:

$$\frac{\Gamma_1 \mid x : \tau_1 \vdash_{\text{M}} e \Rightarrow \tau_2 \dashv \Gamma_2 \qquad \tau = \begin{cases} \tau_1 \to \tau_2 & \text{if } \Gamma_1 = \Gamma_2; \\ \tau_1 \multimap \tau_2 & \text{otherwise.} \end{cases}}{\Gamma_1 \vdash_{\text{M}} \lambda x : \tau_1.\ e \Rightarrow \tau \dashv \Gamma_2}\ (\text{func}\Rightarrow^{\text{bi}}_{\text{M}})$$

$$\frac{\tau_x = \text{useAnno}(\tau_1, \chi) \qquad \Gamma_1 \mid x : \tau_x \vdash_{\text{M}} e \Leftarrow \tau_2 \dashv \Gamma_2 \qquad \tau = \begin{cases} \tau_1 \multimap \tau_2 & \text{if } \Gamma_1 \neq \Gamma_2; \\ \tau_1 \multimap \tau_2 \text{ or } \tau_1 \to \tau_2 & \text{otherwise.} \end{cases}}{\Gamma_1 \vdash_{\text{M}} \lambda \chi.\ e \Leftarrow \tau \dashv \Gamma_2}\ (\text{func}\Leftarrow^{\text{bi}}_{\text{M}})$$

In essence, these rules present the solution to exercise 3.5.4. To define these rules, we make use of the following operation:

**Definition 4.3.2.** Let the partial function useAnno$(\tau, \chi)$ restrict the given type by the annotatable variable. If the annotatable variable is of the form $x$, we do nothing. If of the form $x : \tau_x$, then $\tau$ must act like $\tau_x$. In other words, we have:

$$\text{useAnno}(\tau, \chi) := \begin{cases} \tau_x & \text{if } \chi = x : \tau_x \text{ and } \tau \ <: \ \tau_x \\ \tau & \text{if } \chi = x \end{cases}$$

We have this consistency check because of the following philosophy: type annotations are like assertions, so should always tell the truth. If we know a certain type annotation is wrong, we must abort type checking, which is why we have the subsumption check. The reason we return $\tau_x$ instead of $\tau$ in the top case, is because again, type annotations must tell the truth. We do not want accidentally to use some property of $\tau$, even though we state that the argument should only behave like $\tau_x$.

Also note in the checking rule for functions, that we allow the variable to be annotated. This seems redundant, since we have that case already in the inference rule, but it ensures that we can still typecheck a function with an annotatable variable, even when its body can only be checked, and not inferred.

Recursive function rules rec$\Rightarrow^{\text{bi}}_{\text{M}}$ and rec$\Leftarrow^{\text{bi}}_{\text{M}}$ can always check the body, since the full function type is always available, either through the function variable annotation or through the checking input.

Constants can always be inferred, so the translation to const.$^{\text{bi}}_{\text{M}}$ is evident. Likewise, the rule arith.$^{\text{bi}}_{\text{M}}$ only needs an inference variation (and can even use checking in the premise), since we know they only work on natural numbers.

Pairs need two variants: one inference variant pair$\Rightarrow^{\text{bi}}_{\text{M}}$ where both subexpressions must also be inferred, and one checking variant pair$\Leftarrow^{\text{bi}}_{\text{M}}$ where both subexpresisons can be checked.

Likewise, let-expressions also have two variants — let var$\Rightarrow^{\text{bi}}_{\text{M}}$ and let var$\Leftarrow^{\text{bi}}_{\text{M}}$ for single variables, and let pair$\Rightarrow^{\text{bi}}_{\text{M}}$ and let pair$\Leftarrow^{\text{bi}}_{\text{M}}$ for variable pairs. However, here we need to always infer the first expression, since this information cannot be retrieved from the output type. Let-expressions do not use annotatable variables in the binding, because we can already put type annotations on the first expression. Adding support for annotatable variables there would thus be redundant, and increase complexity of the ruleset.

### 4.3.2. Booleans

The if-statements if 1$\Rightarrow^{\text{bi}}_{\text{M}}$, if 2$\Rightarrow^{\text{bi}}_{\text{M}}$ and if$\Leftarrow^{\text{bi}}_{\text{M}}$ are trivial, but have two inference rules. This is because we only need to infer one of the branches, and can then use the result to check the other branch. However, we don't know which of the two branches may infer, so we have one rule per possibility.

The comparison and ordering rules comp. 1$^{\text{bi}}_{\text{M}}$, comp. 2$^{\text{bi}}_{\text{M}}$, order 1$^{\text{bi}}_{\text{M}}$ and order 2$^{\text{bi}}_{\text{M}}$ have the same situation with inference. However, they do not need a checking rule, since they always return the type $\mathbf{B}$, so each category only has the two inference rules.

The logic rule logic$^{\text{bi}}_{\text{M}}$ is virtually identical to the arithmetic rule arith.$^{\text{bi}}_{\text{M}}$, and the not-rule not$^{\text{bi}}_{\text{M}}$ has the same concept, but is a unary operation instead of a binary one.

### 4.3.3. Choice

The bidirectionalisation of the labeled types into choice$\Rightarrow^{\text{bi}}_{\text{M}}$ and choice$\Leftarrow^{\text{bi}}_{\text{M}}$ is simple; it is a like the pair rules, but using one subexpression instead of two.

However, the match rules are the complete opposite. They are arguably the most complex rules in this chapter, so we go through them slowly, one by one. Recall the algorithmic match rule:

$$\frac{\Gamma_1 \vdash_{\text{M}} e : \{l : \ \tau_l\}_{l \in I} \dashv \Gamma_2 \qquad \forall_{l \in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\text{M}} e_l : \tau \dashv \Gamma_3]}{\Gamma_1 \vdash_{\text{M}} \textbf{match } e \textbf{ with } \{l : \ x_l \mapsto e_l\}_{l \in I} : \tau \dashv \Gamma_3} \ (\text{match}^{\text{alg}}_{\text{M}})$$

We start off with the simplest translation, a checking match with variable annotations in every case:

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \{l:\ \tau_l\}_{l\in I} \dashv \Gamma_2 \qquad \forall_{l\in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l:\ (x_l : \tau_l) \mapsto e_l\}_{l\in I} \Leftarrow \tau \dashv \Gamma_3}\ (\text{match anno}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

To get to an inference rule, we must get the type $\tau$ from one of the cases. Therefore, the inference rule picks one such case arbitrarily to infer (premise 1), and then checks the other cases against the inferred type chosen in premise 3 (read the premises in reading order):

$$\frac{\begin{array}{cc}\{l'\} \subseteq I & \Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \{l:\ \tau_l\}_{l\in I} \dashv \Gamma_2 \\ \Gamma_2 \mid x_{l'} : \tau_{l'} \vdash_{\mathrm{M}} e_{l'} \Rightarrow \tau \dashv \Gamma_3 & \forall_{l\in I\setminus\{l'\}}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l:\ (x_l : \tau_l) \mapsto e_l\}_{l\in I} \Rightarrow \tau \dashv \Gamma_3}\ (\text{match anno}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

We now go back to the checking rule, but for match-expressions that may not be annotated in each case. The rule has five premises:

$$\frac{\begin{array}{ccc}\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \{l:\ \tau'_l\}_{l\in I'} \dashv \Gamma_2 & & I' \subseteq I \\ \forall_{l\in I'}[\tau_l = \text{useAnno}(\tau'_l, \chi_l)] & \forall_{l\in I\setminus I'}[\chi_l = x_l : \tau_l] & \forall_{l\in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l:\ \chi_l \mapsto e_l\}_{l\in I} \Leftarrow \tau \dashv \Gamma_3}\ (\text{match}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

In this rule, we cannot be sure that we have all the information to check $e$ with. Therefore, we must infer it now (premise 1), and then verify that the returned type is consistent with the set of labels $I$ (premise 2) and with any type annotations (premise 3). For cases/labels whose argument types are not in the inferred type, we must retrieve the type from the annotated variable (premise 4). Checking the return types for each case is the same as in other rules (premise 5).

Lastly, we get the corresponding inference rule, which is the same as the checking rule, but – just as the fully annotated inference rule – where one case gets arbitrarily chosen (premise 2) and gets inferred (premise 5) to get type $\tau$:

$$\frac{\begin{array}{cccc}\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \{l:\ \tau'_l\}_{l\in I'} \dashv \Gamma_2 & \{l'\} \subseteq I \quad I' \subseteq I & & \forall_{l\in I'}[\tau_l = \text{useAnno}(\tau'_l, \chi_l)] \\ \forall_{l\in I\setminus I'}[\chi_l = x_l : \tau_l] & \Gamma_2 \mid x_{l'} : \tau_{l'} \vdash_{\mathrm{M}} e_{l'} \Rightarrow \tau \dashv \Gamma_3 & & \forall_{l\in I\setminus\{l'\}}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l:\ \chi_l \mapsto e_l\}_{l\in I} \Rightarrow \tau \dashv \Gamma_3}\ (\text{match}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

## 4.3.4. Concurrency

There is only one fork rule $\text{fork}^{\text{bi}}_{\text{M}}$, since full type information is always available due to the global type annotation.

The receive rules $\text{recv. l.}{\Rightarrow}^{\text{bi}}_{\text{M}}$, $\text{recv. l.}{\Leftarrow}^{\text{bi}}_{\text{M}}$, $\text{recv. unl.}{\Rightarrow}^{\text{bi}}_{\text{M}}$ and $\text{recv. unl.}{\Leftarrow}^{\text{bi}}_{\text{M}}$ are much like the rules for labeled types.

The send rules $\text{send l.}{\Rightarrow}^{\text{bi}}_{\text{M}}$, $\text{send l.}{\Leftarrow}^{\text{bi}}_{\text{M}}$, $\text{send unl.}{\Rightarrow}^{\text{bi}}_{\text{M}}$ and $\text{send unl.}{\Leftarrow}^{\text{bi}}_{\text{M}}$ uses the same strategy as in the function application rules. Namely, in the send rules, the checking versions are sufficiently alternative to the inference versions, but are not strictly stronger.

The close rule $\text{close}^{\text{bi}}_{\text{M}}$ is a trivial rule.

Lastly, redirection is tricky again. Recall the algorithmic redirection rule:

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \pi(L) \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{redirect}_L[\pi](e) : L \dashv \Gamma_2}\ (\text{redirect}^{\text{alg}}_{\text{M}})$$

Here, we needed the local type annotation because $\pi$ is not injective, meaning it does not necessarily have an inverse $\pi^{-1}$. However, for bidirectional typing, we can make two rules, one inference rule for when $\pi$ is injective, and one checking rule for when it might not:

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow L \dashv \Gamma_2 \qquad \text{injective}(\pi)}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{redirect}[\pi](e) \Rightarrow \pi^{-1}(L) \dashv \Gamma_2}\ (\text{redirect}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \pi(L) \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{redirect}[\pi](e) \Leftarrow L \dashv \Gamma_2}\ (\text{redirect}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

Chapter 5.

# Implementation details of MPGVR

In this chapter, we focus on the design of MPGVR. For the full source code, see [22]. Firstly, in section 5.1 we discuss some core dependencies on which the program is built. These include monads from MTL and from `monad-par`, and a custom `Store` monad based on MTL's `State` monad.

Then, we look at how the type checker is implemented in section 5.2. For this, the bidirectional typing rules and corresponding code snippets are put together, to show how they are translated. In this section, we also concern ourselves with some of the intricacies, like choosing when to do subsumption.

In section 5.3 we look at the implementation of the interpreter. Since the code style is much like the type checker, we focus less on showing how the operational semantics are translated, and more on how the MPGV concurrency is implemented.

Parsing is discussed in section 5.4. The parser uses a custom parser combinator, and implements the grammar from appendix B.

Lastly, the REPL combines all parts of the implementation, which is discussed in section 5.5. Here, the script syntax is discussed, and we show that this syntax does not break MPGV semantics. Also, differences between running user input and running scripts are explained.

## 5.1. Monads

Before we look at the MPGV implementation, we must become familiar with the framework on which the implementation is built. The core of the framework consists of the following monad systems:

- the common `MonadError` system from MTL, for error handling;
- a custom `MonadStore` system, for holding the context and evaluation state of an MPGV program;
- the `ParIVar` and `ParFuture` systems from `monad-par` [16], for concurrency.

### 5.1.1. The MTL

The Monad Transformer Library (MTL) is a popular Haskell library[1] that implements some common monad transformers, like `Except`/`ExceptT`, and provides a framework for one to write their own monad transformers. The MPGV interpreter uses this library extensively.

In fact, the custom `Store` monad (and related types and functions) are nothing more but a specialised version of MTL's `State` features. The API of the `Store` monad therefore also tries to resemble that of `State`. The type of state that `Store` holds is fixed to the following type:

```haskell
type StoreContents c = (Map Var (Value c), Map Var Type, Map TypeName Type, Map
    TypeName GType)
```

From left to right, the state holds evaluation information (values of variables), context information (types of variables), type aliases and global type aliases. Here, the type variable `c` holds *channel* data, i.e. holds data on communication between threads. We made it a type variable for abstraction reasons. This way, the monad has no dependency on the concurrency model used, which improves versatility and maintainability.

The reason we made a `Store` monad, and didn't just wrap around the existing `State` monad, is to make sure the store can be added to any already stateful computation. This means no change to the content structure of the existing `State` is needed.

### 5.1.2. monad-par

For concurrency, we use the `monad-par` library[2] by Marlow et al. [16] This library uses monads to allow concurrent programming. Just like MTL, the library provides type classes for its monads, e.g. `ParIVar` and `ParFuture`.

After functions are `fork`ed within the monad, communication can be done through single-use channels, for which `monad-par` uses the type name `IVar`. One thread may send a value over this channel variable, and then another thread — accessing the same channel variable — may receive from it. After that, the channel is closed.

However, in MPGV we may send and receive many different values over the same channel. To go around this restriction of `monad-par`, we introduce the following types:

```haskell
newtype IVarChain c = IVarChain (IVar (Value c, Maybe Label, IVarChain c))

data Channel = Channel {
  sendChannels    :: Map Participant (IVarChain Channel),
  receiveChannels :: Map Participant (IVarChain Channel)
}
```

The `Channel` type is the `monad-par`-specific channel type used throughout the interpreter. As you can see, each `Channel` holds many `IVar`s, two for each other participant — one for sending, one for receiving.

The `IVarChain` type is a recursive wrapper for an `IVar`, allowing for many values to be both sent and received. In order to send an MPGV value (which has type `Value c`) through an `IVarChain`, first the `IVar` in it is extracted. After this, a specific value must be sent through this `IVar`, as described by its type argument (`Value c, Maybe Label, IVarChain c`). Respectively, this is a tuple of the value we want to send, the label chosen (only if we did labeled sending), and a newly created `IVar` from which to perform a new send operation. The receiver of this value must then take note of the new `IVar`, and use that one to perform the next receive operation with.

[1] As of publication of this thesis, the project can be found at https://github.com/haskell/mtl.
[2] As of publication of this thesis, the project can be found at https://github.com/simonmar/monad-par.

Note that not every value can be sent through an `IVar`. Sendable values must be instances of `NFData` (normal form data), which are values that can be forced to evaluate strictly. This ensures that computation is always done in the correct thread. The `Value`, `Channel` and the `IVarChain` types are all instances of this, to ensure that we can send MPGV values, including MPGV channels themselves.

Even though `IVar` channels are bidirectional, we must have a separate map for send-channels and one for receive-channels. This is because in MPGV, two threads can send data to eachother at the same time. Therefore, they could both be using the same channel variable for sending, which breaks the chain. By separating the channel chains into these two maps, we ensure that one thread will always be in charge of sending, and thus extending the chain, and that one keeps reading/receiving from that chain.

## 5.2. Type checker

The type checker implements the bidirectional typing rules from appendix A. This is accomplished through two functions, which mimic the checking and inference typing judgements specifically:

```
checkExpressionType :: (MonadStore c m, MonadError String m) => Expression -> Type ->
    m ()
```

```
typeofExpression :: (MonadStore c m, MonadError String m) => Expression -> m Type
```

In the checking function, we want the type to check against, and we return unit on success, or an error describing what constraint failed. In the inference function, the type is returned instead — or an error if a constraint failed. During type checking, there is no need yet for parallelism, so we do not require a concurrency monad here.

To get familiar with the checker, consider the following code snippets and their corresponding bidirectional typing rules:

```
typeofExpression (ENot e) = checkExpressionType e TBool >> return TBool

typeofExpression (EAnno e t) = checkExpressionType e t >> return t

typeofExpression EUnit = return TUnit

typeofExpression (EBool _) = return TBool

typeofExpression (ENat _) = return TNat

typeofExpression (EVar x) = do
  mt <- lookupType x
  case mt of
    Nothing -> throwError $ "Undefined variable " ++ show x
    Just t  -> do
      cond <- typeIsCopy t
      if cond then
        return t
```

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \tau \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} e : \tau \Rightarrow \tau \dashv \Gamma_2}\ (\mathrm{anno}^{\mathrm{bi}}_{\mathrm{M}}) \qquad \frac{}{\Gamma \vdash_{\mathrm{M}} \mathrm{constant} \Rightarrow \mathrm{type} \dashv \Gamma}\ (\mathrm{const.}^{\mathrm{bi}}_{\mathrm{M}})$$

$$\frac{\Gamma(x) = \tau \qquad \mathrm{copy}(\tau)}{\Gamma \vdash_{\mathrm{M}} x \Rightarrow \tau \dashv \Gamma}\ (\mathrm{var\ copy}^{\mathrm{bi}}_{\mathrm{M}}) \qquad \frac{\Gamma(x) = \tau \qquad \mathrm{linear}(\tau)}{\Gamma \vdash_{\mathrm{M}} x \Rightarrow \tau \dashv \Gamma - x}\ (\mathrm{var\ linear}^{\mathrm{bi}}_{\mathrm{M}})$$

The annotation code forwards the given type to the checking function, as per the semantics of $\mathrm{anno}^{\mathrm{bi}}_{\mathrm{M}}$.

The rule $\mathrm{const.}^{\mathrm{bi}}_{\mathrm{M}}$ is not implemented as a schema, but is instead implemented explicitly for each instance. We did this for efficiency (no need for lookup tables or storing data in the expression itself) and to ensure that nobody can accidentally create a schema instantiation that shouldn't be instantiated.

The variable rules $\mathrm{var\ copy}^{\mathrm{bi}}_{\mathrm{M}}$ and $\mathrm{var\ linear}^{\mathrm{bi}}_{\mathrm{M}}$ are implemented as one function, since the code does a case-distinction on the form of the input expression. Failure of the variable lookup from `lookupType` corresponds with failure of the constraint $\Gamma(x) = \tau$. Since there are no other rules for expressions of the form $x$, on failure we may throw an error stating the variable could not be found. On success, we check whether the type is copy or linear, and modify the context according to the corresponding typing rules, before returning the type.

Thanks to Haskell's monadic notation, the functions read very sequentially, just as we have to do with the bidirectional typing rules. Therefore, most of the inference and checking code should be clear, especially when having the typing rules on hand.

### 5.2.1. Intricacies

Even though the checking and inference functions look innocent, there is a lot that must happen to properly translate the bidirectional typing rules into code. A lot of problems have to do with structural differences. Therefore, we abstract them away by writing utility functions that fix the differences between the typing rules and the code. In this subsection, we explain the most important and non-evident functions.

### Subsumption

The subsumption rule is implemented as the following function:

```haskell
checkExpressionTypeViaTypeof :: (MonadStore c m, MonadError String m) => Expression
    -> Type -> m ()
checkExpressionTypeViaTypeof e t2 = do
  t1 <- typeofExpression e
  cond <- subsumes t1 t2
  if cond then
    return ()
  else
    throwError $ "Expected type " ++ show t2 ++ ", got type " ++ show t1
```

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \tau_1 \ <:\ \tau_2}{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \tau_2 \dashv \Gamma_2}\ (\mathrm{subsumes}^{\mathrm{bi}}_{\mathrm{M}})$$

There is nothing particularly interesting about the snippet itself, but the circumstances in which the function is used is the tricky part. Namely, with the typing rules, subsumption can be applied on every checking step. However, it would make no sense to try and perform subsumption every time a checking step fails, since for most rules, a failing checking step implies a failing inference step.

This is why we have subsumption as a separate function. We use this function in two scenarios during type checking. One scenario is when there is no checking rule, and we must use subsumption to get to the inference rule:

```haskell
checkExpressionType e t = checkExpressionTypeViaTypeof e t
```

The other scenario is during rules like $\mathrm{app}{\Leftarrow}^{\mathrm{bi}}_{\mathrm{M}}$, where we may have to try both checking and inferring:

```haskell
1 checkExpressionType e@(EApp e1 e2) t2 = checkFallbackTypeof
2   (wrapError "In function argument" (typeofExpression e2))
3   (\t1 -> checkExpressionType e1 $ TFunc False t1 t2)
4   e t2
```

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_2 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_1 \Leftarrow \tau_1 \multimap \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} e_1\ e_2 \Leftarrow \tau_2 \dashv \Gamma_3}\ (\mathrm{app}{\Leftarrow}^{\mathrm{bi}}_{\mathrm{M}})$$

In that case, we must first determine whether the inference of the subargument works (line 2). If it works, we can continue checking, using the inferred result (line 3). If the initial inference doesn't work, we must forward to the inference rule instead. If the inference rule fails as well, we still return the original error from the checking step, to show the user that a checking step can be performed. This can be seen in the function that generalises this procedure:

```haskell
checkFallbackTypeof :: (MonadStore c m, MonadError String m)
                    => m a -> (a -> m ()) -> Expression -> Type -> m ()
checkFallbackTypeof f g e t = do
  ea <- (Right <$> f)
        `catchError` (return . Left)
  case ea of
    Right a   -> g a
    Left  err -> checkExpressionTypeViaTypeof e t
                 `catchError` const (throwError err)
```

### Shadowing

Shadowing a variable requires a complex set of steps. Below, we show both the code and the mathematical equivalence from definition 3.4.2.

```haskell
shadowVarType :: (MonadStore c m, MonadError String m) => Var -> Maybe Type -> m a ->
     m a
shadowVarType x t f = do
  t' <- lookupType x
  alterType x t
  a <- f
  cond <- used x
  if cond then do
    alterType x t'
    return a
  else
    throwError $ "Linear variable " ++ show x ++ " has not been used"
  where
    used :: (MonadStore c m, MonadError String m) => Var -> m Bool
    used x = do
      mt <- lookupType x
      case mt of
        Just t  -> typeIsCopy t
        Nothing -> return True
```

$$\begin{aligned}\Gamma_1 \mid x : \tau_x \vdash e : \tau \dashv \Gamma_2 \iff & \exists_{\Gamma_3}[\Gamma_1 \leftarrow (x, \tau_x) \vdash e : \tau \dashv \Gamma_3 \\ & \wedge \text{used}(x, \Gamma_3) \\ & \wedge \Gamma_2 = \Gamma_3 \leftarrow (x, \Gamma_1(x))].\end{aligned}$$

The function requires a `Maybe Type`, since a value of `Nothing` means to remove the variable temporarily. This is to keep consistency with the `alterType` function used, which acts like the $\leftarrow$ operator. The variable `f` holds the deduction to execute while the variable is shadowed.

This function is one in the family of “wrapper” functions. Another wrapper used is for error handling, to show which subexpression(s) we went into. The best example to show how these wrappers are used is in the single variable let-expression:

```
checkExpressionType (ELetVar x e1 e2) t2 = do
  t1 <- wrapError "In let-binding" $ typeofExpression e1
  wrapError "In let-body" . shadowVarType x (Just t1) $ checkExpressionType e2 t2
```

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \mid x : \tau_1 \vdash_{\text{M}} e_2 \Leftarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} \textbf{let } x = e_1 \textbf{ in } e_2 \Leftarrow \tau_2 \dashv \Gamma_3} (\text{let var}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

### Type aliases

For convenience, the interpreter supports type aliases, much like a non-recursive version of Haskell's `type` keyword. The interpreter also allows computation of local type through projection of a global type. However, these features are not supported by the typing rules. Therefore, we have to apply some kind of conversion step. We have done this using the following function:

```
expandType :: (MonadStore c m, MonadError String m) => Type -> m Type
expandType (TName   tn              ) = getNamedType tn >>= expandType
expandType (TLocal  (LTName    tn   )) = getNamedLType tn >>= expandType . TLocal
expandType (TLocal  (LTProject gt p)) = TLocal <$> projectGType p gt
expandType t                          = return t
```

This function looks at any type names or projections, and puts the type into weak head normal form, i.e. a form where the first constructor is not a type name or projection, but where any subtypes may still contain them. Then, any time we must perform pattern matching on a type that may not be in weak head normal form yet, we first have to put it in weak head normal form using this function.

The reason we don't expand the entire type and its subtypes in one go, is it can quickly become inefficient if done too often; we would have to check the entire tree each time, since we don't know if there's an unexpanded part somewhere. Also, using a weak head normal form, error messages can then use type names as much as possible, since they haven't been removed from the type yet. With this we follow Alexandrescu's recommendation [1], ensuring that we don't get an error message fiasco like with C++ template errors.

Note that `getNamedType` and `getNamedLType` may return another type alias or projection, so we must recursively expand. The `projectGType` function on the other hand does return a type in weak head normal form.

## 5.2.2. Closing remarks

The type checker has been shown to work as expected for a number of example cases. In fact, all examples from chapter 2 get through the type checker, and return the correct values without any runtime errors. However, this does not consistute as a formal proof. A full proof that the type checker has been implemented correctly is left to future work. Therefore, we give it as a conjecture.

**Conjecture 5.2.1.** The MPGV type checker is sound and complete with respect to the bidirectional typing rules.

## 5.3. Interpreter

The interpreter is based around the following function that evaluates an expression:

```haskell
evaluateExpression :: (PC.ParIVar P.IVar m, PC.ParFuture P.IVar m, MonadStore Channel
    m)
                   => Expression -> m (Value Channel)
```

This function requires concurrency functionality, and access to the variable store.

Notably, this function does not use the `Except` monad. This is because error handling within a concurrently running process is rather tricky to do, and may have to work very differently depending on how the concurrency monad is implemented. For example, aborting processes in a coroutine environment is much easier than when each concurrent function is a separate process.

For the interpreter, this means that any runtime error will abort the entire program. However, this is not a major issue, since the typechecker is supposed to catch most cases where a runtime error is thrown. The one exception to this is division-by-zero errors.

We now show some basic cases that this function evaluates:

```haskell
evaluateExpression (ENot e) = do
  v <- evaluateExpression e
  case v of
    VBool b -> return . VBool $ not b
    _       -> error $ "Expected boolean value in logical NOT, got value " ++ show v

evaluateExpression (EAnno e _) = evaluateExpression e

evaluateExpression EUnit = return VUnit

evaluateExpression (EBool b) = return $ VBool b

evaluateExpression (ENat n) = return $ VNat n

evaluateExpression (EVar x) = do
  mv <- lookupValue x
  case mv of
    Nothing -> error $ "Undefined variable " ++ show x
    Just v  -> return v
```

$$\frac{E \vdash_{\mathrm{M}} e \mapsto b}{E \vdash_{\mathrm{M}} \textbf{not}\ e \mapsto \neg b}\ (\mathrm{not}_{\mathrm{M}}) \qquad \frac{E \vdash_{\mathrm{M}} e \mapsto v}{E \vdash_{\mathrm{M}} e : \tau \mapsto v}\ (\mathrm{anno}_{\mathrm{M}})$$

$$\frac{}{E \vdash_{\mathrm{M}} \text{constant} \mapsto \text{value}}\ (\mathrm{const}_{\mathrm{M}}) \qquad \frac{E(x) = v}{E \vdash_{\mathrm{M}} x \mapsto v}\ (\mathrm{var}_{\mathrm{M}})$$

Note that the operational semantics do not have an output evaluation. This is because now we are not concerned with linearity anymore; the type checker already ensures that linear variables are not used twice. Therefore, no case within `evaluateExpression` shall indudce a net change in the store — of course shadowing modifies the store, but changes are reverted later.

On the topic of shadowing, it is again implemented with a wrapper function:

```haskell
shadowVarValue :: (PC.ParIVar P.IVar m, PC.ParFuture P.IVar m, MonadStore Channel m)
               => Var -> Maybe (Value Channel) -> m a -> m a
shadowVarValue x v f = do
  v' <- lookupValue x
  alterValue x v
  a <- f
  alterValue x v'
  return a
```

For the interpreter, we also have a conjecture on the correctness:

**Conjecture 5.3.1.** The MPGV interpreter is sound and complete with respect to the operational semantics.

### 5.3.1. Concurrency

Concurrency-related operations are done whenever a fork-, send-, receive-, redirect- or close-expression is being evaluated. These functions implement MPGV-style concurrency by making specific calls to the monad-par monad.

Recall the channel datatype:

```
newtype IVarChain c = IVarChain (IVar (Value c, Maybe Label, IVarChain c))

data Channel = Channel {
  sendChannels    :: Map Participant (IVarChain Channel),
  receiveChannels :: Map Participant (IVarChain Channel)
}
```

For forking, first we create maps of IVar channels, wrapped in an IVarChain to make them multi-use. These values will fill the sendChannels part for each thread's MPGV channel. Then, each value is copied into the receiving counterpart, filling the receiveChannels. Lastly, the actual threads are created, using the following helper function:

```
    spawnThread sends receives p (x, e) = PC.fork $ do
      alterValue x . Just $ makeChannel sends receives p
      v <- evaluateExpression e
      case v of
        VUnit -> return ()
        _      -> error $ "Expected to return unit at end of thread, received " ++
    show v
```

Whenever a thread is forked, monad-par will take the current Store and copy it over to the forked thread. Upon completion of the thread, the Store is discarded. This is done by making Store part of the SplittableState class. With these semantics, stores are easy to work with during concurrency and are deterministic. Note that this is another major reason for the operational semantics to not have an output context; if it had one, the semantics of discarding the State would become problematic.

Sending and receiving goes in the fashion described before: the sending side continues the IVarChain, and the receiving side consumes part of the chain. So for the sending part, we have:

```
          i' <- IVarChain <$> PC.new
          PC.put i (v, ml, i')
          return $ VChannel (Channel (Map.insert p' i' sends) receives) r
```

And for the receiving part, we have:

```
          (v, ml, i') <- PC.get i
          let channel = VChannel (Channel sends (Map.insert p' i' receives)) r
          case ml of
            Just l -> return . VChoice l $ VPair v channel
            Nothing -> return $ VPair v channel
```

Notable variable names are p', the (possibly redirected) participant, and i', the continuation of the IVarChain. In both snippets, we must modify the channel so that it contains the proper continuation of the IVarChain.

For redirect, we simply add the given redirection to the existing one in the channel. We must take care, however, to prioritise redirections from the given mapping.

Lastly, closing is extremely simple: we can just evaluate the expression, then return unit. This is beacuse checking that a thread ends with unit is part of fork's responsibility, and can be seen in its helper function. The actual closing of the threads is done automatically by monad-par once the helper function completes.

## 5.4. Parser

The parser is implemented using a custom parser combinator `Parser`. This parser tries to go through all possible parsing combinations. Once it found a combination that transforms all parts of the string into a syntactically correct MPGV program, it returns said program. If instead it couldn't find any combination, an error is returned for the combination that came furthest. The reason behind this, is that the combination that came furthest is most likely to be the intended combination.

Just like with the type checker, errors may be wrapped to display more contextual information. Also, the position from which parsing could not go further is displayed with the error whenever possible, to give a better idea of the syntax mistake made.

As an example of how the combinator works, consider the `Type` parser:

```
1  pType, pType2, pType3 :: Bool -> Parser Type
2  
3  pType withTypeNames = (TFunc False <$> pType2 withTypeNames <* pStringToken "-%" <*> pType withTypeNames)
4                    <|> (TFunc True  <$> pType2 withTypeNames <* pStringToken "->" <*> pType withTypeNames)
5                    <|> pType2 withTypeNames
6  
7  pType2 withTypeNames = pFoldl (pType3 withTypeNames) [("*", TPair)]
8  
9  pType3 withTypeNames = (TUnit <$ pStringToken "()")
10                    <|> (TBool <$ pStringToken "B")
11                    <|> (TNat <$ pStringToken "N")
12                    <|> (TChoice <$> pMatch ((,) <$> pLabelToken <* pStringToken ":" <*> pType withTypeNames))
13                    <|> (TLocal <$> pLType False withTypeNames)
14                        -- Must come after TLocal so as to allow global type projections with type names
15                    <|> (if withTypeNames then TName <$> pTypeNameToken else pError "Invalid type")
16                    <|> pParenToken (pType withTypeNames)
17                    <|> pError "Invalid type"
```

This code follows the MPGV grammar as described in appendix B. To ensure that the parser doesn't end up in an infinite loop, and to allow for operator precedences, we have multiple parsing layers. Each layer implements different operations on the same precedence level, and consumes subtypes on a lower level. Only the last layer may loop back to the first layer.

For function types (see lines 3 and 4), the parser combinator can already evaluate in the correct order. When parsing pairs, however, the combinator goes in the wrong direction. For this, the `pFoldl` parser exists, which implements a `foldl`-like parsing operation on a set of binary expressions. In this case, the only binary expression on this layer is the pair expression. The implementation of this parser can be seen below:

```
pFoldl :: Parser a -> [(String, a -> a -> a)] -> Parser a
pFoldl p binOps = do
  base <- p
  others <- many . asum . map (\(i, (op, _)) -> (,) i <$ pStringToken op <*> p) $ zip [0..] binOps
  return $ foldl (\e1 (i, e2) -> (snd $ binOps !! i) e1 e2) base others
```

What this function does, is first parse an expression of a lower level. After that, it must read one of the binary operation tokens, and then read another lower level expression. After parsing as many as possible, we can fold them together in the correct direction by applying `foldl` with the correct binary operation constructors. An example of this parser can be seen in the first expression layer:

```
pExpression = pFoldl pExpression2 [("&&", EAnd), ("||", EOr)]
```

## 5.5. REPL

The REPL ties the whole implementation together, by performing parsing, type checking and interpreting in a constant interactive loop. The REPL can parse both individual lines of user input and entire script files. For this, some new syntax is introduced.

### 5.5.1. Metacommands

A user can type certain *metacommands* into the interface. These commands give information about the state of the REPL, or allow the user to perform special actions. For example, `.sv` shows the values of all variables in the store. The command `.q` quits the program. If there are still linear variables in play, the interpreter will display a warning asking if you really want to quit. The command `.t e` shows the type of the expression `e`, without performing evaluation of `e`. Notably, it also undoes any changes made to the store during said type checking. For a full list of commands, use `.h`.

### 5.5.2. Script syntax

The script syntax, used for both user input and for scripts, is described in appendix B. This syntax consists of printing, variable assignment, type aliasing and script loading functionalities. For convenience, the relevant part of the appendix is given below:

```
-- S   ::=                    ; script
--          SL                    ; ending statement
--          SL\nS                 ; intermediate statement
--
-- SL  ::=                    ; script line
--         print e                ; printed expression
--         e                      ; printed expression (only in REPL)
--         let ax=e               ; variable assignment
--         ax=e                   ; variable assignment (only in REPL)
--         type TN=T              ; type alias (type names disallowed in T)
--         gtype TN=GT            ; global type alias (type names disallowed in GT)
--         import "s"             ; load script at the given path (no escape codes
    allowed)
```

#### Printing

To evaluate an expression `e` and print its value, you can use `print e`. For this, it is required that `e` does not return a linear value. Without the restriction, the REPL could be consuming an important linear variable, like a channel.

Chaining individual `print` statements together can be modeled with the existing MPGV syntax. Namely, the code

```
print e1
print e2
```

can be modeled as the expression

```
let unusedVariable = e1 in e2
```

where the value of `unusedVariable` is printed to the screen after evaluating `e1`. From the typing rule of let-expressions, we see that evaluating `e1` can be done before knowing about `e2`. Therefore, the program can be read and evaluated statement-by-statement.

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \mid x : \tau_1 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{let } x = e_1 \textbf{ in } e_2 \Leftarrow \tau_2 \dashv \Gamma_3} \text{(let var}{\Leftarrow}^{\mathrm{bi}}_{\mathrm{M}}\text{)}$$

When evaluating user input in the REPL, the `print` keyword may be omitted. The reason it is required in scripts is because it resolves ambiguity between a print-statement and function application. Since user input is parsed on a per-statement basis, this ambiguity does not exist for user input.

### Variable assignment

In order to work with variables, the syntax `let x = e` or `let x::T = e` is used. In this case, `e` may return a linear value. Namely, the linear value can be used up by placing the corresponding variable into later statements. This type of statement can also be modeled using existing MPGV syntax by using let-in-expressions, but where the variable may be used. Just as with printing, the `let` keyword may be omitted when evaluating user input, but not when evaluating scripts.

### Type aliasing

The statements `type TN = T` and `gtype TN = GT` introduce type aliases. These aliases can then be used in later expressions. Since these are just aliases, MPGV semantics are not changed, because the type checker can just expand the types whenever needed. Type aliases may not be redefined, however, to ensure that lazy expansion of type aliases in previous expressions does not suddenly expand to the wrong newly created definition. The exception to this is when the type alias is redefined to exactly the same type. This is done to make it possible to `import` scripts that have type aliases multiple times, without getting a redefinition-error. To summarise, we have the following behaviour:

```
type CustomType = N->N -- OK, initial definition
type CustomType = N->N -- OK, redefinition to same type
type CustomType = N-%N -- ERROR, redefinition to different type
```

### Script loading

To load the contents of a script, and execute them as if the script were typed out straight into the REPL, the `import "some/path"` syntax is used. This is much like the `#include "some/path"` syntax used in languages like C and C++, except here the script is expanded at runtime instead of at compile time. So we immediately see that loading scripts this way doesn't break the guarantees of MPGV. Namely, loading scripts is nothing more than if the script contents were placed inline at the import-statement instead, forming a bigger script.

## 5.5.3. Error handling

To allow for exceptions to be handled in the REPL during parsing and type checking, the following helper function is used:

```
withPrintError :: (ParIVar IVar m, MonadStore Channel m, MonadIO m)
               => StoreT Channel (Except String) a -> (a -> m ()) -> m () -> m ()
withPrintError m f g = do
  s <- getStore
  case runExcept $ runStoreT m s of
    Left e -> do
      printError e
      g
    Right (a, s') -> do
      putStore s'
      f a
```

This function runs in the same monad as the interpreter, but creates an environment with a store and with error handling, in which the type checker and parser can be run. The store is moved to the new environment. The function `m` is the monadic function to run within said environment. If the function throws an error, we print the error to the screen and continue with executing `g`. If the function returns a value `a` successfully, we update the store and execute `f a`.

## 5.5.4. Cleanup

When going from interpreting statements to type checking new statements, we must perform cleanup. This is because the interpreter does not effectively change the store, meaning that linear variables are not removed when used. However, we would like the `.sv` command to only show linear variables that may still be used. Therefore, we look at what variables get deleted during type checking of an expression, and remove those after interpreting the corresponding expression. This is not a problem when executing scripts, since they get type checked in one go before being evaluated in one go. However, when evaluating user input, we cannot do this. Therefore, the REPL contains this code for evaluating statements from user input:

```
    AScriptLine str -> withPrintError
      (liftEither $ parseEval (pScriptLine True) str)
      (\sl -> checkScriptLine sl
        (\sl' -> do
          runScriptLine sl'
          ts <- getTypes
          -- Make sure we remove variables that have been thrown away
          -- during type checking, but haven't during evaluation
          modifyValues $ (flip Map.intersection) ts
          repl)
        repl)
      repl
```

Here, we first parse the input, going back to the REPL on error. Otherwise, we check the type of the statement, again going to the REPL on error. Otherwise, we evaluate the statement, collect the list of variables that still exist after type checking, and mirror that list on the interpreter side.

Chapter 6.

# Related work

## Session type systems

There exist many different implementations for session types, each with different features. Below, we show a table comparing a variety of session type systems with our MPGV implementation.

| **in paper**<br>**name** | —<br>MPGV | [17]<br>AlgST | [4]<br>Exceptional GV | [13]<br>Rusty Variation | [15]<br>MultiCrusty | [11]<br>Gradual GV | [14]<br>Priority Sesh |
|---|---|---|---|---|---|---|---|
| **written in/for** | Haskell | Haskell | Links | Rust | Rust | — | Linear Haskell |
| **implementation** | interpreted | interpreted | library | library | library | — | library |
| **type inference** | bidirectional | bidirectional | inherited | inherited | inherited | — | inherited |
| **channel typing** | linear | linear | linear | affine | affine | linear, gradual | linear |
| **deadlock-free** | ✓ | × | ✓ | ✓ | ✓ | × | ✓ |
| **multiparty** | ✓ | × | × | × | ✓ | × | × |
| **cancellation** | × | × | ✓ | ✓ | ✓ | × | ✓ |
| **dynamic threads** | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ |

In the next paragraphs, we describe some of these entries in more detail, and explain parts that may be unclear.

## AlgST and bidirectional typing

The AlgST implementation [17] is most similar to our MPGV implementation. It is also written in Haskell, implements a session type system, and uses bidirectional type checking. However, it uses binary session types (opposed to multiparty session types), and the kind of session type framework is different.

AlgST introduces session types based on algebraic protocols. It is an effort to create concise syntax for recursive session types, like exists with parameterised algebraic datatypes. For example, suppose we wish to make a protocol where we send an arbitrary amount of natural numbers. In AlgST, it would be written as `protocol RecN = End | Cons N RecN`. This is similar to how you would write a `List` type in Haskell. On the other hand, in MPGV you would write `gtype RecN = [0->1]{end:(). End; cons:N. RecN}`[1]. Since MPGV's syntax shares similarities with Haskell, such algebraic protocol syntax could be an interesting addition to the language.

Our implementation uses a bidirectionalisation recipe that allows for inference and checking rules of the same expression type (see section 3.5). The reason for this, is that checking rules can sometimes take different routes than inference rules can. Having these different routes increases the expressivity of the type system, and enables us to remove many more annotations.

On the other hand, AlgST only implements part of the recipe; only inference rules are provided, the exception being the subsumption rule. This means that the typing rules allow for subsumption to occur, but that annotations are still required in all places; only in checking rules do we get enough type information to omit annotations. To give a better idea of where this places AlgST's type checking, consider the following example:

```
let copyFunc :: () -> () = \u. u
let f :: (() -% ()) -> () = \linFunc. linFunc ()
print f copyFunc
```

Recall that `-%` indicates a linear function. In MPGV, this code would type check, since (1) the argument types in `copyFunc` and `f` can be inferred from the function's types, and (2) since subsumption allows copy functions to be passed as linear arguments. However, in AlgST, only (2) is implemented. Therefore, the code will not type check in AlgST, since the argument types cannot be inferred; they need to be annotated, like this:

```
let copyFunc :: () -> () = \u :: (). u
let f :: (() -% ()) -> () = \linFunc :: () -% (). linFunc ()
print f copyFunc
```

[1]MPGVR does not currently have recursive types, but this syntax would be the most logical if it were added.

Now the code will compile with the AlgST strategy, since the code is fully annotated. Note that the bidirectional strategy used for AlgST still provides value over normal algorithmic typing rules, because of subsumption. Without subsumption, the above code would still not compile, since `f` would require a linear function, and would not accept a copy function.

**MultiCrusty and cancellation** MultiCrusty [15] is a Rust library implementing AMPST, an affine multiparty session type calculus [15]. The type checking capabilities therefore are inherited from the Rust compiler used. The library supports multiparty types, and has an affine type system — instead of a linear one.

An affine type system relaxes the restriction that each variable must be used. This means that channels may or may not be used, which is an issue for deadlock-free protocols. To combat this issue, MultiCrusty has "cancellation" systems in place.

Cancellation is a technique where we detect dropped channels and proceed to terminate the protocol safely if a dropped channel is found. After termination, the threads that are still alive can start a recovery procedure, for example one where they restart the protocol. Cancellation solves the problem of deadlocks from dropped channels, since an aborted protocol cannot deadlock.

Aborting the protocol after detecting a dropped channel can be considered a rather crude way of preventing deadlocks. Preferably, we could provide recovery procedures within the protocol. This way, we have more scenarios in which we can recover and continue or take an alternative route, without having to restart the entire protocol or set up a new protocol. MultiCrusty supports this more fine-grained form of recovery, by having `try P catch Q` and `cancel(c).P` constructs in the calculus. Within the library, these are implemented through the `attempt! {...} catch {...}` macro and `cancel()` function. Below, we show some examples of MultiCrusty cancellation, inspired by the examples in [15].

```
match s.recv() {
    Ok((v, s)) => M
    Err(e) => {
        // Other threads get cancellation errors when continuing the protocol
        cancel(s);
        // Stop this thread
        panic!("Error: {:?}", e)
    }
}
```

```
attempt! {
    s.recv();
    get_video(); // Errors
} catch (e) {
    //
    // Recover this step of the protocol
    //
}
```

Even though MPGV theoretically does not have runtime errors (apart from division-by-zero), having a cancellation system in place is useful, and could also be considered an improvement to MPGV. For example, the operating system may decide to kill a thread, possibly causing the other threads to deadlock. This is a real-world situation that the theory does not account for. Using this cancellation system would weaken the guarantees of the language ("the protocol always succeeds"), but in practice could be worth it.

The last row of the table may need some explaining. There can be multiple protocols active at the same time in MultiCrusty. However, these must be merged into one central spot using the `create_fork_interleave!` macro. Therefore, protocols cannot dynamically be run, in contrast to GV-based languages like MPGV.

**Gradual GV and gradual types** Gradual GV [11] is a calculus that implements gradual linear session types. These are linear session types, for which the type may be checked at runtime instead of being checked statically. The motivation behind Gradual GV is that multiple parties can communicate even if one uses a statically typed language, while the other uses a dynamically typed language[11]; with Gradual GV, one can switch between static typing and dynamic typing at will.

**Concurrency libraries** Our MPGV implementation uses the `monad-par` library for concurrency. However, there are many more ways to do concurrency in Haskell. We now elaborate on why we made the choice for `monad-par`.

One important feature of `monad-par` is that we can choose the type of thread scheduling to use. Depending on the work done with the MPGV implementation, this choice can become very valuable. For example, one might choose to play with deterministic scheduling and its consequences over non-deterministic scheduling.

When using a system like `MVar` for message passing and the `Control.Concurrent`'s `forkIO` for spawning threads, we lose the ability to provide our own scheduling. The `async` library also suffers from this issue.

With `Control.Parallel`, we do get the benefit of multiple scheduling strategies. However, this library only provides basic parallelism. To make the implementation support message passing, we need full concurrency.

Chapter 7.

# Conclusion

In this thesis, we introduced an implementation of the MPGV language. This language was devised by Jacobs et al. [12], and unifies two major branches of session types: GV [5, 6, 23] for deadlock freedom, and MPST [9, 10] for multiparty session types.

The implementation is written in Haskell, and includes parser, type checking and interpreter modules. The code is based on the algorithmic formalisations described in this thesis. As seen in chapter 5, many parallels can be drawn between these formalisations and the actual code written. Of course, this correspondence does not automatically make the code correct, but it does make it easier to verify the correctness of the code. Formal verification is part of future work (see section 7.1).

A notable formalisation implemented is the algorithmic shadowing structure from section 3.4. The formalisation distills the necessary steps needed to shadow a variable, and combines these steps seamlessly with the algorithmic judgements. This reduces the chance of errors made during shadowing.

The bidirectional typing recipe from Dunfield and Krishnaswami. [3] has proven useful as well. As seen in section 4.3, translation of MPGV's regular typing rules — as described in [12] — into algorithmic bidirectional rules succeeded. It has eliminated the need for many annotations, and generalised the different annotations; these resulting annotation-expressions and annotatable variables provide a much more powerful and flexible way to provide annotations to an MPGV program.

Certain syntactical improvements have been made to the MPGV language, compared to the original specification in [12]. These have been showcased in chapter 2, together with a translation from mathematical syntax into ASCII syntax. The most notable of these improvements include shorthand syntax for channel operations (e.g. `c.send(1)`) and unlabeled sending and receiving (e.g. `!N. End` as opposed to `!{Label: N. End}`). Some of these changes were able to be defined in terms of the original specification. Others have been added to the language by means of new typing rules and operational semantics.

It was also possible to write a REPL environment. This same environment also allows MPGV to function as an embedded language; one can manage the environment, including variables and calls to scripts, all from within their own Haskell code by hooking into provided monads.

## 7.1. Future work

**Recursive types** The implementation conforms with nearly all the specifications as laid out by Jacobs et al. [12] However, the feature of recursive types is still missing. It is left as future work since this feature requires major changes to certain existing systems.

The most notable change needed is that comparison between types becomes non-trivial. Bookkeeping needs to be added to prevent infinite loops during comparisons such as equality and subsumption.

Once recursive types are added, type aliases may want to be reworked as well, to support references to types defined later, allowing for cyclic definitions. Such feature may require the choice to make types nominal, i.e. two types with the same implementation but different names are not equal anymore. One may want to implement AlgST's syntax for types, since it supports recursive and nominal types. [17]

**Verification** The MPGV implementation was verified by means of examples, some of which can be seen in chapter 2. However, formal verification of soundness and completeness of the typing rules — for example by means of a proof assistant — has yet to be done. As a baseline, the Coq code from [12] that formalises the MPGV language can be used.

Also, the effeciency of the type checker has not been thouroughly tested. The fact that the examples compile in reasonable time implies that the interpreter is fairly efficient. However, there may be certain combinations in which the time to type check the code might not scale well. This should be analysed, and any deficiencies should be corrected in the typing rules, to ensure that MPGV can be practically type checked.

**Error handling** Lastly, error handling is still lacking in the implementation. When parsing and type checking error, this can be recovered from. However, runtime errors of MPGV code make the whole implementation abort. This is not a major issue, since due to the safety guarantees from type checking there shouldn't be many runtime errors. The only one that can originate from inside of MPGV itself is division-by-zero; any other errors come from components like the operating system, for example during thread spawning. One might want to fix this problem by implementing cancellation, as done in related languages [4, 13, 15].

**Convenience features** Another feature to add is to allow for participant names like "Alice" and "Bob" instead of numeric identifiers like 0 and 1. It was used in the examples by Jacobs et al. [12] to increase readability.

The syntax for match-clauses and fork-clauses is cumbersome. Namely, suppose we have a function that implements one of the participants to fork:

```
let f :: SomeChannel -> () =
  \c.
    do something;
    c.close()
```

Then, the fork-clause must accept the parameter like this: `fork<GT>(c. f c)`. We would much rather accept functions, though, so that we can type `fork<GT>(f)`, and then by extension allow writing them inline as `fork<GT>(\c. do something; c.close())`.

# Bibliography

[1] A. Alexandrescu. Better template error messages. *C/C++ Users Journal*, 17(3):37–47, 1999. ISSN: 1075-2838. Publisher: Miller Freeman, Inc.

[2] H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[3] J. Dunfield and N. Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):1–38, June 30, 2022. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3450952. URL: https://dl.acm.org/doi/10.1145/3450952 (visited on 11/24/2022).

[4] S. Fowler, S. Lindley, J. G. Morris, and S. Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3:28:1–28:29, POPL, 2019. DOI: 10.1145/3290341. URL: https://doi.org/10.1145/3290341.

[5] S. Gay and V. Vasconcelos. Asynchronous functional session types. *IEEE Transactions on Reliability*, May 2007.

[6] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. DOI: 10.1017/S0956796809990268. URL: https://doi.org/10.1017/S0956796809990268.

[7] K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. DOI: 10.1007/3-540-57208-2_35. URL: https://doi.org/10.1007/3-540-57208-2%5C_35.

[8] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. DOI: 10.1007/BFb0053567. URL: https://doi.org/10.1007/BFb0053567.

[9] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA. Association for Computing Machinery, 2008. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472. URL: https://doi.org/10.1145/1328438.1328472. event-place: San Francisco, California, USA.

[10] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):1–67, Mar. 30, 2016. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/2827695. URL: https://dl.acm.org/doi/10.1145/2827695 (visited on 09/26/2022).

[11] A. Igarashi, P. Thiemann, Y. Tsuda, V. T. Vasconcelos, and P. Wadler. Gradual session types. *Journal of Functional Programming*, 29:e17, 2019. DOI: 10.1017/S0956796819000169. Publisher: Cambridge University Press.

[12] J. Jacobs, S. Balzer, and R. Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6:466–495, ICFP, Aug. 29, 2022. ISSN: 2475-1421. DOI: 10.1145/3547638. URL: https://dl.acm.org/doi/10.1145/3547638 (visited on 09/20/2022).

[13] W. Kokke. Rusty Variation: deadlock-free sessions with failure in rust. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Ex-*

*perience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *EPTCS*, pages 48–60, 2019. DOI: 10.4204/EPTCS.304.4. URL: https://doi.org/10.4204/EPTCS.304.4.

[14] W. Kokke and O. Dardha. Deadlock-free session types in linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell.* ICFP '21: 26th ACM SIGPLAN International Conference on Functional Programming, pages 1–13, Virtual Republic of Korea. ACM, Aug. 18, 2021. ISBN: 978-1-4503-8615-9. DOI: 10.1145/3471874.3472979. URL: https://dl.acm.org/doi/10.1145/3471874.3472979 (visited on 09/26/2022).

[15] N. Lagaillardie, R. Neykova, and N. Yoshida. Stay safe under panic: affine Rust programming with multiparty session types. *CoRR*, abs/2204.13464, 2022. DOI: 10.48550/arXiv.2204.13464. arXiv: 2204.13464. URL: https://doi.org/10.48550/arXiv.2204.13464.

[16] S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In K. Claessen, editor, *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 71–82. ACM, 2011. DOI: 10.1145/2034675.2034685. URL: https://doi.org/10.1145/2034675.2034685.

[17] A. Mordido, J. Spaderna, P. Thiemann, and V. T. Vasconcelos. Parameterized algebraic protocols. *CoRR*, abs/2304.03764, 2023. DOI: 10.48550/arXiv.2304.03764. arXiv: 2304.03764. URL: https://doi.org/10.48550/arXiv.2304.03764.

[18] B. C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2004. ISBN: 0-262-16228-8. URL: https://www.cis.upenn.edu/~bcpierce/attapl/ (visited on 03/07/2023).

[19] K. Rupp. 42 years of microprocessor trend data. 42 Years of Microprocessor Trend Data | Karl Rupp. Feb. 15, 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ (visited on 05/04/2023).

[20] C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In pages 35–50, July 2007. ISBN: 978-3-540-73594-6. DOI: 10.1007/978-3-540-73595-3_4.

[21] C. Urban and M. Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding.* MERLIN05: Mechanized Reasoning about Languages with Variable Binding 2005 Workshop, pages 25–32, Tallinn Estonia. ACM, Sept. 27, 2005. ISBN: 978-1-59593-072-9. DOI: 10.1145/1088454.1088458. URL: https://dl.acm.org/doi/10.1145/1088454.1088458 (visited on 02/21/2023).

[22] M. F. van Kessel. MPGVR: the artifact described in "Achieving multiparty deadlock-free communication through linear typing", version 1.0.0.0, 2023. URL: https://doi.org/10.5281/zenodo.8130945.

[23] P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA. Association for Computing Machinery, 2012. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364568. URL: https://doi.org/10.1145/2364527.2364568. event-place: Copenhagen, Denmark.

Chapter A.

# Typing rules of MPGV

## A.1. Regular typing rules

### Leaves

$$\frac{\mathrm{copy}(\Gamma)}{\Gamma \cup \{(x,\tau)\} \vdash_{\mathrm{M}} x : \tau} \ (\mathrm{var}_{\mathrm{M}}^{\mathrm{reg}})$$

For the constant/type pairs $((), \mathbf{1})$, $(\mathrm{True}, \mathbf{B})$ and $(\mathrm{False}, \mathbf{B})$, and for natural number constants $(n, \mathbf{N})$, we derive typing rules according to the following schema:

$$\frac{\mathrm{copy}(\Gamma)}{\Gamma \vdash_{\mathrm{M}} \mathrm{constant} : \mathrm{type}} \ (\mathrm{const.}_{\mathrm{M}}^{\mathrm{reg}})$$

### Boolean logic

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \mathbf{B} \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \qquad \Gamma_2 \vdash_{\mathrm{M}} e_3 : \tau}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau} \ (\mathrm{if}_{\mathrm{M}}^{\mathrm{reg}})$$

$$\frac{\Gamma \vdash_{\mathrm{M}} e : \mathbf{B}}{\Gamma \vdash_{\mathrm{M}} \mathbf{not}\ e : \mathbf{B}} \ (\mathrm{not}_{\mathrm{M}}^{\mathrm{reg}})$$

For the binary comparison operations `==` and `/=`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \qquad \mathrm{eqComp}(\tau)}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B}} \ (\mathrm{comp.}_{\mathrm{M}}^{\mathrm{reg}})$$

For the binary ordering operations `<`, `<=`, `>` and `>=`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \qquad \mathrm{orderable}(\tau)}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B}} \ (\mathrm{order}_{\mathrm{M}}^{\mathrm{reg}})$$

For the binary logic operations `&&` and `||`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma \vdash_{\mathrm{M}} e_1 : \mathbf{B} \qquad \Gamma(\mathbb{0}) \vdash_{\mathrm{M}} e_2 : \mathbf{B}}{\Gamma \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B}} \ (\mathrm{logic}_{\mathrm{M}}^{\mathrm{reg}})$$

### Functions

$$\frac{\mathrm{copy}(\Gamma) \qquad \Gamma \cup \{(x,\tau_1)\} \vdash_{\mathrm{M}} e : \tau_2}{\Gamma \vdash_{\mathrm{M}} \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2} \ (\mathrm{func\ copy}_{\mathrm{M}}^{\mathrm{reg}}) \qquad \frac{\Gamma \cup \{(x,\tau_1)\} \vdash_{\mathrm{M}} e : \tau_2}{\Gamma \vdash_{\mathrm{M}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2} \ (\mathrm{func\ linear}_{\mathrm{M}}^{\mathrm{reg}})$$

$$\frac{\mathrm{copy}(\Gamma) \qquad \Gamma \cup \{(f, \tau_1 \to \tau_2), (x,\tau_1)\} \vdash_{\mathrm{M}} e : \tau_2}{\Gamma \vdash_{\mathrm{M}} \mathbf{rec}\ f_{\tau_1 \to \tau_2}\ x_{\tau_1}.\ e : \tau_1 \to \tau_2} \ (\mathrm{rec}_{\mathrm{M}}^{\mathrm{reg}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_1}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} e_1\ e_2 : \tau_2} \ (\mathrm{app}_{\mathrm{M}}^{\mathrm{reg}})$$

### Pairs

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_2}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} (e_1, e_2) : \tau_1 \times \tau_2} \; (\mathrm{pair}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2 \cup \{(x_1, \tau_1), (x_2, \tau_2)\} \vdash_{\mathrm{M}} e_2 : \tau_3}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{let } (x_1, x_2) = e_1 \textbf{ in } e_2 : \tau_3} \; (\text{let pair}^{\mathrm{reg}}_{\mathrm{M}})$$

### Choice

$$\frac{\Gamma \vdash_{\mathrm{M}} e : \tau_l \qquad l \in I}{\Gamma \vdash_{\mathrm{M}} \langle l : e \rangle_{\{l' :\; \tau_{l'}\}_{l' \in I}} : \{l' :\; \tau_{l'}\}_{l' \in I}} \; (\mathrm{choice}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e : \{l :\; \tau_l\}_{l \in I} \qquad \forall_{l \in I} [\Gamma_2 \cup \{(x_l, \tau_l)\} \vdash_{\mathrm{M}} e_l : \tau]}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{match } e \textbf{ with } \{l :\; x_l \mapsto e_l\}_{l \in I} : \tau} \; (\mathrm{match}^{\mathrm{reg}}_{\mathrm{M}})$$

### Concurrency

$$\frac{\Gamma_1 \perp \cdots \perp \Gamma_n \qquad \mathrm{participants}(G) \subseteq \{0, \ldots, n\} \qquad \forall_{1 \le i \le n} [\Gamma_i \cup \{(x_i, G \downarrow i)\} \vdash_{\mathrm{M}} e_i : \mathbf{1}]}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash_{\mathrm{M}} \textbf{fork}\langle G \rangle (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) : G \downarrow 0} \; (\mathrm{fork}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : ![p]\{l' :\; \tau_{l'}.\, L_{l'}\}_{l' \in I} \qquad l \in I \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_l}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, l, e_2) : L_l} \; (\text{send l.}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : ![p]\tau.\, L \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, e_2) : L} \; (\text{send unl.}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma \vdash_{\mathrm{M}} e : ?[p]\{l :\; \tau_l.\, L_l\}_{l \in I}}{\Gamma \vdash_{\mathrm{M}} \textbf{receive}[p](e) : \{l :\; \tau_l \times L_l\}_{l \in I}} \; (\text{recv. l.}^{\mathrm{reg}}_{\mathrm{M}}) \qquad \frac{\Gamma \vdash_{\mathrm{M}} e : ?[p]\tau.\, L}{\Gamma \vdash_{\mathrm{M}} \textbf{receive}[p](e) : \tau \times L} \; (\text{recv. unl.}^{\mathrm{reg}}_{\mathrm{M}})$$

$$\frac{\Gamma \vdash_{\mathrm{M}} e : \pi(L)}{\Gamma \vdash_{\mathrm{M}} \textbf{redirect}_L[\pi](e) : L} \; (\mathrm{redirect}^{\mathrm{reg}}_{\mathrm{M}}) \qquad \frac{\Gamma \vdash_{\mathrm{M}} e : \mathrm{End}}{\Gamma \vdash_{\mathrm{M}} \textbf{close}(e) : \mathbf{1}} \; (\mathrm{close}^{\mathrm{reg}}_{\mathrm{M}})$$

### Other

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \qquad \Gamma_2 \cup \{(x, \tau_1)\} \vdash_{\mathrm{M}} e_2 : \tau_2}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \; (\text{let var}^{\mathrm{reg}}_{\mathrm{M}})$$

For the binary arithmetic operations `+`, `-`, `*` and `/`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{M}} e_1 : \mathbf{N} \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \mathbf{N}}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{N}} \; (\text{arith.}^{\mathrm{reg}}_{\mathrm{M}})$$

## A.2. Algorithmic typing rules

### Leaves

$$\frac{\Gamma(x) = \tau \qquad \mathrm{copy}(\tau)}{\Gamma \vdash_{\mathrm{M}} x : \tau \dashv \Gamma} \; (\text{var copy}^{\mathrm{alg}}_{\mathrm{M}}) \qquad \frac{\Gamma(x) = \tau \qquad \mathrm{linear}(\tau)}{\Gamma \vdash_{\mathrm{M}} x : \tau \dashv \Gamma - x} \; (\text{var linear}^{\mathrm{alg}}_{\mathrm{M}})$$

For the constant/type pairs $((), \mathbf{1})$, $(\mathrm{True}, \mathbf{B})$ and $(\mathrm{False}, \mathbf{B})$, and for natural number constants $(n, \mathbf{N})$, we derive typing rules according to the following schema:

$$\frac{}{\Gamma \vdash_{\mathrm{M}} \mathrm{constant} : \mathrm{type} \dashv \Gamma} \; (\text{const.}^{\mathrm{alg}}_{\mathrm{M}})$$

### Boolean logic

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_3 : \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \dashv \Gamma_3}\ (\text{if}^{\text{alg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \mathbf{B} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{not}\ e : \mathbf{B} \dashv \Gamma_2}\ (\text{not}^{\text{alg}}_{\mathrm{M}})$$

For the binary comparison operations `==` and `/=`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \dashv \Gamma_3 \qquad \text{eqComp}(\tau)}{\Gamma_1 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B} \dashv \Gamma_3}\ (\text{comp.}^{\text{alg}}_{\mathrm{M}})$$

For the binary ordering operations `<`, `<=`, `>` and `>=`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \dashv \Gamma_3 \qquad \text{orderable}(\tau)}{\Gamma_1 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B} \dashv \Gamma_3}\ (\text{order}^{\text{alg}}_{\mathrm{M}})$$

For the binary logic operations `&&` and `||`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2(\mathbb{0}) \vdash_{\mathrm{M}} e_2 : \mathbf{B} \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{B} \dashv \Gamma_2}\ (\text{logic}^{\text{alg}}_{\mathrm{M}})$$

### Functions

$$\frac{\Gamma_1(\mathbb{0}) \mid x : \tau_1 \vdash_{\mathrm{M}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \lambda x_{\tau_1}.\ e : \tau_1 \to \tau_2 \dashv \Gamma_1}\ (\text{func copy}^{\text{alg}}_{\mathrm{M}}) \qquad \frac{\Gamma_1 \mid x : \tau_1 \vdash_{\mathrm{M}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \lambda^{\mathbb{1}} x_{\tau_1}.\ e : \tau_1 \multimap \tau_2 \dashv \Gamma_2}\ (\text{func linear}^{\text{alg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1(\mathbb{0}) \mid f : \tau_1 \to \tau_2 \mid x : \tau_1 \vdash_{\mathrm{M}} e : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{rec}\ f_{\tau_1 \to \tau_2}\ x_{\tau_1}.\ e : \tau_1 \to \tau_2 \dashv \Gamma_1}\ (\text{rec}^{\text{alg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau \dashv \Gamma_2 \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_1 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} e_1\ e_2 : \tau_2 \dashv \Gamma_3}\ (\text{app}^{\text{alg}}_{\mathrm{M}})$$

### Pairs

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} (e_1, e_2) : \tau_1 \times \tau_2 \dashv \Gamma_3}\ (\text{pair}^{\text{alg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \times \tau_2 \dashv \Gamma_2 \qquad \Gamma_2 \mid x_1 : \tau_1 \mid x_2 : \tau_2 \vdash_{\mathrm{M}} e_2 : \tau_3 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau_3 \dashv \Gamma_3}\ (\text{let pair}^{\text{alg}}_{\mathrm{M}})$$

### Choice

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \tau_l \dashv \Gamma_2 \qquad l \in I}{\Gamma_1 \vdash_{\mathrm{M}} \langle l : e \rangle_{\{l' :\ \tau_{l'}\}_{l' \in I}} : \{l' :\ \tau_{l'}\}_{l' \in I} \dashv \Gamma_2}\ (\text{choice}^{\text{alg}}_{\mathrm{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \{l :\ \tau_l\}_{l \in I} \dashv \Gamma_2 \qquad \forall_{l \in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l : \tau \dashv \Gamma_3]}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{match}\ e\ \mathbf{with}\ \{l :\ x_l \mapsto e_l\}_{l \in I} : \tau \dashv \Gamma_3}\ (\text{match}^{\text{alg}}_{\mathrm{M}})$$

### Concurrency

$$\frac{\text{participants}(G) \subseteq \{0, \dots, n\} \qquad \forall_{1 \leq i \leq n}[\Gamma_i \mid x_i : G \downarrow i \vdash_{\mathrm{M}} e_i : \mathbf{1} \dashv \Gamma_{i+1}]}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{fork}\langle G \rangle(x_1 \mapsto e_1, \dots, x_n \mapsto e_n) : G \downarrow 0 \dashv \Gamma_{n+1}}\ (\text{fork}^{\text{alg}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : ![p]\{l' : \ \tau_{l'}.\ L_{l'}\}_{l' \in I} \dashv \Gamma_2 \qquad l \in I \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau_l \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{send}[p](e_1, l, e_2) : L_l \dashv \Gamma_3}\ (\text{send l.}^{\text{alg}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : ![p]\tau.\ L \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{send}[p](e_1, e_2) : L \dashv \Gamma_3}\ (\text{send unl.}^{\text{alg}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : ?[p]\{l : \ \tau_l.\ L_l\}_{l \in I} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{receive}[p](e) : \{l : \ \tau_l \times L_l\}_{l \in I} \dashv \Gamma_2}\ (\text{recv. l.}^{\text{alg}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : ?[p]\tau.\ L \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{receive}[p](e) : \tau \times L \dashv \Gamma_2}\ (\text{recv. unl.}^{\text{alg}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e : \pi(L) \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{redirect}_L[\pi](e) : L \dashv \Gamma_2}\ (\text{redirect}^{\text{alg}}_{\text{M}}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{M}} e : \text{End} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{close}(e) : \mathbf{1} \dashv \Gamma_2}\ (\text{close}^{\text{alg}}_{\text{M}})$$

### Other

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \mid x : \tau_1 \vdash_{\mathrm{M}} e_2 : \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \dashv \Gamma_3}\ (\text{let var}^{\text{alg}}_{\text{M}})$$

For the binary arithmetic operations `+`, `-`, `*` and `/`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 : \mathbf{N} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 : \mathbf{N} \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} e_1 \odot e_2 : \mathbf{N} \dashv \Gamma_3}\ (\text{arith.}^{\text{alg}}_{\text{M}})$$

## A.3. Bidirectional typing rules

### Basic rules

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \tau \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} e : \tau \Rightarrow \tau \dashv \Gamma_2}\ (\text{anno}^{\text{bi}}_{\text{M}}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \tau_1 \ <: \ \tau_2}{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \tau_2 \dashv \Gamma_2}\ (\text{subsumes}^{\text{bi}}_{\text{M}})$$

### Leaves

$$\frac{\Gamma(x) = \tau \qquad \text{copy}(\tau)}{\Gamma \vdash_{\mathrm{M}} x \Rightarrow \tau \dashv \Gamma}\ (\text{var copy}^{\text{bi}}_{\text{M}}) \qquad \frac{\Gamma(x) = \tau \qquad \text{linear}(\tau)}{\Gamma \vdash_{\mathrm{M}} x \Rightarrow \tau \dashv \Gamma - x}\ (\text{var linear}^{\text{bi}}_{\text{M}})$$

For the constant/type pairs $((), \mathbf{1})$, $(\text{True}, \mathbf{B})$ and $(\text{False}, \mathbf{B})$, and for natural number constants $(n, \mathbf{N})$, we derive typing rules according to the following schema:

$$\frac{}{\Gamma \vdash_{\mathrm{M}} \text{constant} \Rightarrow \text{type} \dashv \Gamma}\ (\text{const.}^{\text{bi}}_{\text{M}})$$

### Boolean logic

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Leftarrow \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Rightarrow \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_3 \Leftarrow \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow \tau \dashv \Gamma_3}\ (\text{if } 1{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Leftarrow \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_3 \Rightarrow \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow \tau \dashv \Gamma_3}\ (\text{if } 2{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Leftarrow \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_3 \Leftarrow \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Leftarrow \tau \dashv \Gamma_3}\ (\text{if}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \mathbf{B} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \mathbf{not}\ e \Rightarrow \mathbf{B} \dashv \Gamma_2}\ (\text{not}^{\text{alg}}_{\text{M}})$$

For the binary comparison operations `==` and `/=`, we derive typing rules according to the following two schemas (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_2 \Leftarrow \tau \dashv \Gamma_3 \qquad \text{eqComp}(\tau)}{\Gamma_1 \vdash_{\text{M}} e_1 \odot e_2 \Rightarrow \mathbf{B} \dashv \Gamma_3}\ (\text{comp. } 1^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_2 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_1 \Leftarrow \tau \dashv \Gamma_3 \qquad \text{eqComp}(\tau)}{\Gamma_1 \vdash_{\text{M}} e_1 \odot e_2 \Rightarrow \mathbf{B} \dashv \Gamma_3}\ (\text{comp. } 2^{\text{bi}}_{\text{M}})$$

For the binary ordering operations `<`, `<=`, `>` and `>=`, we derive typing rules according to the following two schemas (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_2 \Leftarrow \tau \dashv \Gamma_3 \qquad \text{orderable}(\tau)}{\Gamma_1 \vdash_{\text{M}} e_1 \odot e_2 \Rightarrow \mathbf{B} \dashv \Gamma_3}\ (\text{order } 1^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_2 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_1 \Leftarrow \tau \dashv \Gamma_3 \qquad \text{orderable}(\tau)}{\Gamma_1 \vdash_{\text{M}} e_1 \odot e_2 \Rightarrow \mathbf{B} \dashv \Gamma_3}\ (\text{order } 2^{\text{bi}}_{\text{M}})$$

For the binary logic operations `&&` and `||`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Leftarrow \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2(\mathbb{0}) \vdash_{\text{M}} e_2 \Leftarrow \mathbf{B} \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} e_1 \odot e_2 \Rightarrow \mathbf{B} \dashv \Gamma_2}\ (\text{logic}^{\text{bi}}_{\text{M}})$$

## Functions

$$\frac{\Gamma_1 \mid x : \tau_1 \vdash_{\text{M}} e \Rightarrow \tau_2 \dashv \Gamma_2 \qquad \tau = \begin{cases} \tau_1 \to \tau_2 & \text{if } \Gamma_1 = \Gamma_2; \\ \tau_1 \multimap \tau_2 & \text{otherwise.} \end{cases}}{\Gamma_1 \vdash_{\text{M}} \lambda x : \tau_1.\ e \Rightarrow \tau \dashv \Gamma_2}\ (\text{func}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\tau_x = \text{useAnno}(\tau_1, \chi) \qquad \Gamma_1 \mid x : \tau_x \vdash_{\text{M}} e \Leftarrow \tau_2 \dashv \Gamma_2 \qquad \tau = \begin{cases} \tau_1 \multimap \tau_2 & \text{if } \Gamma_1 \neq \Gamma_2; \\ \tau_1 \multimap \tau_2 \text{ or } \tau_1 \to \tau_2 & \text{otherwise.} \end{cases}}{\Gamma_1 \vdash_{\text{M}} \lambda \chi.\ e \Leftarrow \tau \dashv \Gamma_2}\ (\text{func}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1(\mathbb{0}) \mid f : \tau_1 \to \tau_2 \mid x : \tau_1 \vdash_{\text{M}} e \Leftarrow \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{M}} \mathbf{rec}\ (f : \tau_1 \to \tau_2)\ x.\ e \Rightarrow \tau_1 \to \tau_2 \dashv \Gamma_1}\ (\text{rec}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\tau_x \to \tau_2' = \text{useAnno}(\tau_1 \to \tau_2, \chi) \qquad \Gamma_1(\mathbb{0}) \mid f : \tau_x \to \tau_2' \mid x : \tau_x \vdash_{\text{M}} e \Leftarrow \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash_{\text{M}} \mathbf{rec}\ \chi\ x.\ e \Leftarrow \tau_1 \to \tau_2 \dashv \Gamma_1}\ (\text{rec}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Rightarrow \tau \dashv \Gamma_2 \qquad \tau \in \{\tau_1 \to \tau_2, \tau_1 \multimap \tau_2\} \qquad \Gamma_2 \vdash_{\text{M}} e_2 \Leftarrow \tau_1 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} e_1\ e_2 \Rightarrow \tau_2 \dashv \Gamma_3}\ (\text{app}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_2 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_1 \Leftarrow \tau_1 \multimap \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} e_1\ e_2 \Leftarrow \tau_2 \dashv \Gamma_3}\ (\text{app}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

## Pairs

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_2 \Rightarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} (e_1, e_2) \Rightarrow \tau_1 \times \tau_2 \dashv \Gamma_3}\ (\text{pair}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e_1 \Leftarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\text{M}} e_2 \Leftarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} (e_1, e_2) \Leftarrow \tau_1 \times \tau_2 \dashv \Gamma_3}\ (\text{pair}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e \Rightarrow \tau_1 \times \tau_2 \dashv \Gamma_2 \qquad \Gamma_2 \mid x_1 : \tau_1 \mid x_2 : \tau_2 \vdash_{\text{M}} e_2 \Rightarrow \tau_3 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau_3 \dashv \Gamma_3}\ (\text{let pair}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\text{M}} e \Rightarrow \tau_1 \times \tau_2 \dashv \Gamma_2 \qquad \Gamma_2 \mid x_1 : \tau_1 \mid x_2 : \tau_2 \vdash_{\text{M}} e_2 \Leftarrow \tau_3 \dashv \Gamma_3}{\Gamma_1 \vdash_{\text{M}} \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \Leftarrow \tau_3 \dashv \Gamma_3}\ (\text{let pair}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

## Choice

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \tau \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \langle l : e\rangle \Rightarrow \{l' :\ \tau\}_{l' \in \{l\}} \dashv \Gamma_2}\ (\text{choice}{\Rightarrow}^{\text{bi}}_{\text{M}}) \qquad \frac{l \in I \qquad \Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \tau_l \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \langle l : e\rangle \Leftarrow \{l' :\ \tau_{l'}\}_{l' \in I} \dashv \Gamma_2}\ (\text{choice}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \{l :\ \tau'_l\}_{l \in I'} \dashv \Gamma_2 \qquad \{l'\} \subseteq I \qquad I' \subseteq I \qquad \forall_{l \in I'}[\tau_l = \text{useAnno}(\tau'_l, \chi_l)] \\ \forall_{l \in I \setminus I'}[\chi_l = x_l : \tau_l] \qquad \Gamma_2 \mid x_{l'} : \tau_{l'} \vdash_{\mathrm{M}} e_{l'} \Rightarrow \tau \dashv \Gamma_3 \qquad \forall_{l \in I \setminus \{l'\}}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l :\ \chi_l \mapsto e_l\}_{l \in I} \Rightarrow \tau \dashv \Gamma_3}\ (\text{match}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow \{l :\ \tau'_l\}_{l \in I'} \dashv \Gamma_2 \qquad I' \subseteq I \\ \forall_{l \in I'}[\tau_l = \text{useAnno}(\tau'_l, \chi_l)] \qquad \forall_{l \in I \setminus I'}[\chi_l = x_l : \tau_l] \qquad \forall_{l \in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l :\ \chi_l \mapsto e_l\}_{l \in I} \Leftarrow \tau \dashv \Gamma_3}\ (\text{match}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\begin{array}{c}\{l'\} \subseteq I \qquad \Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \{l :\ \tau_l\}_{l \in I} \dashv \Gamma_2 \\ \Gamma_2 \mid x_{l'} : \tau_{l'} \vdash_{\mathrm{M}} e_{l'} \Rightarrow \tau \dashv \Gamma_3 \qquad \forall_{l \in I \setminus \{l'\}}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]\end{array}}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l :\ (x_l : \tau_l) \mapsto e_l\}_{l \in I} \Rightarrow \tau \dashv \Gamma_3}\ (\text{match anno}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \{l :\ \tau_l\}_{l \in I} \dashv \Gamma_2 \qquad \forall_{l \in I}[\Gamma_2 \mid x_l : \tau_l \vdash_{\mathrm{M}} e_l \Leftarrow \tau \dashv \Gamma_3]}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{match}\ e\ \textbf{with}\ \{l :\ (x_l : \tau_l) \mapsto e_l\}_{l \in I} \Leftarrow \tau \dashv \Gamma_3}\ (\text{match anno}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

## Concurrency

$$\frac{\text{participants}(G) \subseteq \{0, \ldots, n\} \qquad \forall_{1 \le i \le n}[\Gamma_i \mid x_i : G \downarrow i \vdash_{\mathrm{M}} e_i \Leftarrow \mathbf{1} \dashv \Gamma_{i+1}]}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{fork}\langle G\rangle(x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) \Rightarrow G \downarrow 0 \dashv \Gamma_{n+1}}\ (\text{fork}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Rightarrow {!}[p]\{l' :\ \tau_{l'}.\ L_{l'}\}_{l' \in I} \dashv \Gamma_2 \qquad l \in I \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau_l \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, l, e_2) \Rightarrow L_l \dashv \Gamma_3}\ (\text{send l.}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_2 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_1 \Leftarrow {!}[p]\{l :\ \tau.\ L\} \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, l, e_2) \Leftarrow L \dashv \Gamma_3}\ (\text{send l.}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Rightarrow {!}[p]\tau.\ L \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, e_2) \Rightarrow L \dashv \Gamma_3}\ (\text{send unl.}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_2 \Rightarrow \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_1 \Leftarrow {!}[p]\tau.\ L \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{send}[p](e_1, e_2) \Leftarrow L \dashv \Gamma_3}\ (\text{send unl.}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow {?}[p]\{l :\ \tau_l.\ L_l\}_{l \in I} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{receive}[p](e) \Rightarrow \{l :\ \tau_l \times L_l\}_{l \in I} \dashv \Gamma_2}\ (\text{recv. l.}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow {?}[p]\{l :\ \tau_l.\ L_l\}_{l \in I} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{receive}[p](e) \Leftarrow \{l :\ \tau_l \times L_l\}_{l \in I} \dashv \Gamma_2}\ (\text{recv. l.}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow {?}[p]\tau.\ L \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{receive}[p](e) \Rightarrow \tau \times L \dashv \Gamma_2}\ (\text{recv. unl.}{\Rightarrow}^{\text{bi}}_{\text{M}}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow {?}[p]\tau.\ L \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{receive}[p](e) \Leftarrow \tau \times L \dashv \Gamma_2}\ (\text{recv. unl.}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Rightarrow L \dashv \Gamma_2 \qquad \text{injective}(\pi)}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{redirect}[\pi](e) \Rightarrow \pi^{-1}(L) \dashv \Gamma_2}\ (\text{redirect}{\Rightarrow}^{\text{bi}}_{\text{M}}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \pi(L) \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{redirect}[\pi](e) \Leftarrow L \dashv \Gamma_2}\ (\text{redirect}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e \Leftarrow \text{End} \dashv \Gamma_2}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{close}(e) \Rightarrow \mathbf{1} \dashv \Gamma_2}\ (\text{close}^{\text{bi}}_{\text{M}})$$

## Other

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \mid x : \tau_1 \vdash_{\mathrm{M}} e_2 \Rightarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \tau_2 \dashv \Gamma_3}\ (\text{let var}{\Rightarrow}^{\text{bi}}_{\text{M}})$$

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Rightarrow \tau_1 \dashv \Gamma_2 \qquad \Gamma_2 \mid x : \tau_1 \vdash_{\mathrm{M}} e_2 \Leftarrow \tau_2 \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \tau_2 \dashv \Gamma_3}\ (\text{let var}{\Leftarrow}^{\text{bi}}_{\text{M}})$$

For the binary arithmetic operations `+`, `-`, `*` and `/`, we derive typing rules according to the following schema (replace $\odot$ with the operation):

$$\frac{\Gamma_1 \vdash_{\mathrm{M}} e_1 \Leftarrow \mathbf{N} \dashv \Gamma_2 \qquad \Gamma_2 \vdash_{\mathrm{M}} e_2 \Leftarrow \mathbf{N} \dashv \Gamma_3}{\Gamma_1 \vdash_{\mathrm{M}} e_1 \odot e_2 \Rightarrow \mathbf{N} \dashv \Gamma_3}\ (\text{arith.}^{\text{bi}}_{\text{M}})$$

Chapter B.

# MPGVR grammar

This grammar description is taken from the source code of MPGVR [22]:

```
--------------------------------------------------
-- MPGVR grammar
--------------------------------------------------
-- Whitespace is implicit in this grammar,
-- and allowed between any two tokens.
-- Comments (c) can be placed anywhere where whitespace is allowed.
--
-- Variables (x), labels (l) and type names (TN) may only use the characters a-z, A-Z, 0-9, ' and _.
-- Variables must start with one of a-z or _.
-- Labels and type names must start with one of A-Z.
-- Variables may be shadowed.
--
-- Natural numbers are indicated by the letter n.
-- Participants are also natural numbers, and are indicated by the letter p.
-- Single-line strings of text are indicated by the letter s.
--
-- Reserved words are:
--   B, close, else, False, fork, gtype, import, if, in, let, match, N,
--   not, print, rec, receive, redirect, send, then, True, type, with
--
-- c  ::=                     ; comment
--        --s\n                   ; single line comment
--
-- ax ::=                     ; annotatable variables
--        x                       ; variable
--        x::T                    ; variable with type annotation
--
-- C  ::=                     ; contexts
--        [x:T, ...]              ; variable-type binding
--
-- E  ::=                     ; evaluations
--        [x:v, ...]              ; variable-value binding
--
-- r  ::=                     ; partial redirection function
--        p->p, ...               ; redirect participants
--
-- e  ::=                     ; expressions (precedence 1 (lowest), binds (e.e).e)
--        e&&e2                  ; logical AND
--        e||e2                  ; logical OR
--        e2                     ; other expression
-- e2 ::=                     ; expressions (precedence 2, binds (e.e).e)
--        e2==e3                  ; equality comparison
--        e2/=e3                  ; inequality comparison
--        e2<=e3                  ; less-than-or-equal-to comparison
--        e2>=e3                  ; greater-than-or-equal-to comparison
--        e2<e3                   ; less-than comparison
--        e2>e3                   ; greater-than comparison
--        e3                      ; other expression
-- e3 ::=                   ; expressions (precedence 3, binds (e.e).e)
--        e3+e4                 ; addition
--        e3-e4                 ; proper subtraction
--        e4                    ; other expression
-- e4 ::=                   ; expressions (precedence 4, binds (e.e).e)
--        e4*e5                 ; multiplication
--        e4/e5                 ; division
--        e5                    ; other expression
-- e5 ::=                   ; expressions (precedence 5, binds (e.e).e)
--        e5 e6                 ; function application
--        e6                    ; other expression
-- e6 ::=                   ; expressions (precedence 6 (highest))
--        e7::T                 ; expression with type annotation
--        not e7                ; logical NOT
--        e7                    ; other expression
-- e7 ::=                   ; expression (unambiguous)
--        x                     ; bounded variable
--        ()                    ; unit
--        True                  ; boolean TRUE
--        False                 ; boolean FALSE
--        n                     ; natural number
--        (e,e)                 ; pair
--        <l:e>                 ; labeled expression
--        \ax.e                 ; function
--        rec ax x.e            ; recursive function
--        fork<GT>(x.e, ...)    ; fork with global type annotation
--        send[p](e, e)         ; send message
--        send[p](e, l, e)      ; send message with choice
--        receive[p](e)         ; receive message
--        close(e)              ; close channel
--        redirect[r](e)        ; redirect participants
--        let x=e in e          ; let-binding (var)
--        let (x,x)=e in e      ; let-binding (pair)
--        if e then e else e            ; if-then-else
--        match e with {l:ax.e; ...}    ; label matching
--        (e)                           ; encapsulation
--
-- v   ::=                  ; values
--        ()                    ; unit
--        True                  ; boolean TRUE
--        False                 ; boolean FALSE
--        n                     ; natural number
--        (v,v)                 ; pair
--        <l:v>                 ; labeled value
--        \x.e E                ; function with evaluation
--        rec x x.e E           ; recursive function with evaluation
--        #[p/n,r]              ; channel (as part. no. and part. count)
--        (v)                   ; encapsulation
--
-- T   ::=                  ; types (precedence 1 (lowest), binds e.(e.e))
--        T2-%T                 ; linear function
--        T2->T                 ; unrestricted function
--        T2                    ; other type
-- T2 ::=                   ; types (precedence 2 (highest), binds (e.e).e)
--        T2*T3                 ; pair
--        T3                    ; other type
-- T3 ::=                   ; types (unambiguous)
--        ()                    ; unit
--        B                     ; boolean
--        N                     ; natural number
--        {l:T; ...}            ; labeled type
--        LT                    ; local type
--        TN                    ; type name (disallowed sometimes)
--        (T)                   ; encapsulation
--
-- LT ::=                   ; local types
--         ![p]T.LT                ; send-action
--         ?[p]T.LT                ; receive-action
--         ![p]{l:T.LT; ...}       ; send-action with choice/match type
--         ?[p]{l:T.LT; ...}       ; receive-action with choice/match type
--         GT|n                    ; projection of global type to participant
--         End                     ; end
--         TN                      ; (local) type name (disallowed sometimes)
--
-- GT ::=                      ; global types
--         [p->p]T.GT              ; action
--         [p->p]{l:T.GT; ...}     ; action with choice/match type
--         End                     ; end
--         TN                      ; (global) type name (disallowed sometimes)
--
-- S   ::=                     ; script
--          SL                     ; ending statement
--          SL\nS                  ; intermediate statement
--
-- SL  ::=                     ; script line
--         print e                 ; printed expression
--         e                       ; printed expression (only in REPL)
--         let ax=e                ; variable assignment
--         ax=e                    ; variable assignment (only in REPL)
--         type TN=T               ; type alias (type names disallowed in T)
--         gtype TN=GT             ; global type alias (type names disallowed in GT)
--         import "s"              ; load script at the given path (no escape codes
    allowed)
--
-- Alternative notation for expressions on channels is as follows,
-- where x is a channel variable, and y is a variable:
--   x.send[p](e')         -> send[p](x, e')
--   x.send[p](e'); e      -> let x=send[p](x, e') in e
--   x.send[p](l, e')      -> send[p](x, l, e')
--   x.send[p](l, e'); e -> let x=send[p](x, l, e') in e
--   x.receive[p]()        -> receive[p](x)
--   y=x.receive[p](); e -> let (y,x)=receive[p](x) in e
--   x.close()             -> close(x)
--   x.close(); e          -> let x=close(x) in e
--   x.redirect[r]()       -> redirect[r](x)
--   x.redirect[r](); e   -> let x=redirect[r](x) in e
```