Asynchronous Sessions with Implicit Functions and Messages†

Alex Jeffery
University of Sussex
Email: A.P.Jeffery@sussex.ac.uk

Martin Berger
University of Sussex
Email: M.F.Berger@sussex.ac.uk

***Abstract*—Session types are a well-established approach to ensuring protocol conformance and the absence of communication errors such as deadlocks in message passing systems. Haskell introduced implicit parameters, Scala popularised this feature and recently gave implicit types first-class status, yielding an expressive tool for handling context dependencies in a type-safe yet terse way. We ask: can type-safe implicit functions be generalised from Scala's sequential setting to message passing computation? We answer this question in the affirmative by presenting the first concurrent functional language with implicit message passing. The key idea is to generalise the concept of an implicit function to an *implicit message*, its concurrent analogue. Our language extends Gay and Vasconcelos's calculus of linear types for asynchronous sessions (*LAST*) with implicit functions and messages. We prove the resulting system sound by translation into LAST.**

I. Introduction

***Session types*.** Types classify programs, distinguishing between programs that are guaranteed to exhibit semantic properties of interest, and those that are not. Types for sequential computation are well-established and a core part of industrial software engineering. Behavioural type systems extend types to concurrent, parallel and distributed computation, and are a core activity of contemporary type theoretical research.

Session types, first introduced in [3], [10] are an important example of a behavioural type system for message passing concurrency. Session types classify message passing behaviour at given channels: e.g. if process P first receives an integer and then a boolean on channel x, and finally sends a boolean on x, then this behaviour could be expressed by the session type

```
!Int.?Bool.!Bool.end
```

Here `?T` represents input of a value of type `T`, `!T` means sending a value that has type `T`, while `end` denotes the end of the interaction.

A key notion in session types is that of *duality*, originating in linear logic: processes P and Q can be composed in parallel only when throughout the course of the computation each output of P's is matched by a suitable input of Q's, and vice versa. Session types allow only dual processes to interact. Hence typability guarantees the absence of communication errors such as mismatched communication and deadlocks. A process Q, dual to P above, would have the session type

```
?Int.!Bool.?Bool.end
```

Notice that for each action in P's type, we have the dual action in Q's type, e.g. an output of type `!Int` can be received by an input of type `?Int`.

***Implicit functions*.** Modularity, a core concept in software engineering, is greatly aided by parameterisation of programs. Parameterisation has dual facets: supplying and consuming a parameter. A key tension in large-scale software engineering is between *explicit* (e.g. pure functional programming), and *implicit* parameterisation (e.g. global state). The former enables local reasoning but can lead to repetitive supply of parameters. Here is a simple example of the problem (where <= is the function $\lambda xy.x \leq y$, and α a type):

```
let f x compare : α = ... in
    f 3 (<=)
    ...
    f 17 (<=)
    ...
```

Repeatedly passing functions like <= which are unlikely to change frequently, is tedious, and impedes readability of large code bases. Default parameters are an early proposal for mediating this tension in a type-safe way. The key idea is to annotate function arguments with their default value, to be used whenever an invocation does not supply an argument:

```
let f x (compare = (<=)) : α = ... in
   f 2
   ...
   f 5
   ...
```

The compiler synthesises `f 2 (<=)` from `f 2`, and `f 5 (<=)` from `f 5`. Default parameters have a key disadvantage: the default value is hard-coded at the

†Extended abstract. We thank S. Gay, A. Scalas and V. Vasconcelos for valuable feedback on the present work.

978-1-5386-5541-2/18/$31.00

callee, and cannot be context dependent. Implicit arguments, a strict generalisation of default parameters, were pioneered in Haskell [6], and popularised as well as refined in Scala [7]: they separate the *callee's declaration* that an argument can be elided, from the *caller's choice* of elided values, allowing the latter to be context dependent.

```
let f x (implicit compare) : α = ... in
   let g = ... in
      let implicit cmp = (<=)
         f 2
   ...
   let h = ... in
      let implicit cmp = (>)
         f 5
   ...
```

In this example `f 2` is rewritten as above, but `f 5` becomes `f 5 (>)`, i.e. a different implicit argument is synthesised. The disambiguation between several providers of implicit arguments happens at compile-time using type and scope information. Programs where elided arguments cannot be disambiguated at compile-time are rejected as ill-formed. Hence type-safety is not compromised.

One might ask: can type-safe implicit functions be generalised from Scala's sequential setting to message passing computation? We answer this question in the affirmative by generalising the concept of implicit functions to *implicit messages*. We elaborate on this idea by presenting the first concurrent functional language with implicit message passing: we extend Gay and Vasconcelos's calculus of linear types for asynchronous sessions (*LAST*) [2] with implicit message passing and implicit functions. We argue with several examples that implicit messages provide useful abstractions for programming languages with session types. In particular, repeated rebinding of session names can be omitted.

***Implicit messages*.** The concept of implicit messages has two dual parts:

- Input can be declared implicit, and not be explicitly matched by an output in the dual process.
- At compile time, a suitable output is synthesised, based on type and scope information.

In the following example, we have two processes `p` and `q` running in parallel (we write $\|$ for parallel composition). They initiate a session (denoted by `accept` and `request`) on fresh and dual channels `c` (for messages from `p` to `q`) and `d` (for messages from `q` to `p`), whereafter `p` performs an (implicit) receive and `q` apparently does nothing.

```
⟨ let p =
    let c = accept x in
    let n, c = implicit receive c in c
in p ⟩ ‖ ⟨ let q =
    let ≀ = 10 in
    let d = request x in d
in q ⟩
```

The type system sees the implicit receive in `p` and is able to figure out that a corresponding send must be inserted into `q`. It knows that the channel that the send occurs on is `d` since it is the dual channel to `c` which the `implicit receive` uses. The chosen message is a variable of appropriate type from the implicit scope. The implicit scope can be thought of as a store of variables that are designated as implicit - we make this notion more precise in Section IV. Following [7], we do not give names to implicit variables until after translation, but use ≀ (pronounced 'query') as a placeholder name for all implicit variables. The translation becomes:

```
⟨ let p =
    let c = accept x in
    let n, c = receive c in c
in p ⟩ ‖ ⟨ let q =
    let y = 10 in
    let d = request x in d
    send y d
in q ⟩
```

Here `y` is a fresh variable. This insertion corresponds to adding an implicit variable as an additional argument to an implicit function.

***Elimination of repeated rebinding*.** A well-known problem with the integration of session types and sequential languages is the seeming necessity of repeated rebinding of channel names. The problem is that `send` takes a channel of type $!T.S$ as its second argument, and returns a linear channel of type S. In order for linearity to be respected that channel must be rebound. Consider the process below, typical of LAST programs.

```
miscService :: ⟨S⟩^a → end
miscService ap =
    let    c = accept ap in
    let m, c = receive c in
    let n, c = receive c in
    if pred(m) then
        let c = select l1 c in
        let c = send f(m, n) c in
        let c = send g(m, n, n) c in c
    else
        let c = select l2 c in
        let o, c = receive c in
        let c = send f(n, m) c in
        let c = send g(m, n, o) c in c
```

This redundancy makes programs hard to read. The issue can be addressed in other ways, for example using parameterised monads [1], see also [2, Chapter 7]. Implicit functions and message passing enable a principled and canonical solution: make the channel argument implicit and let the compiler synthesise the missing channel name for rebinding.

The `send` primitive has type $T \to !T.S \to S$. We can use implicit function types to define a new output primitive `send`$^{\wr}$, with type $T \to !T.S \wr\!\to S$, explained

in detail below. The annotation $\wr$ in $!T.S \wr\!\!\rightarrow S$ makes the channel argument implicit, and the returned channel is rebound to the implicit scope by the body of $\texttt{send}^{\wr}$ and is not required elsewhere.

```
send^≀ :: T → !T.S ≀↦ S → U → U
send^≀ m u = let ≀ = send m ≀ in u
```

We can do something similar for `select` and `receive`.

```
select^≀ :: Label → ⊕⟨...l:S,...⟩ ≀↦ S → U → U
select^≀ l u = let ≀ = select l ≀ in u

receive^≀ :: ?T.S ≀↦ T
receive^≀ = let m, ≀ = receive ≀ in m
```

We can rewrite `miscService` above with our new primitives. The resulting code is less repetitive and more terse, hence readable.

```
miscService :: ⟨S⟩^a ≀↦ end
miscService =
    let ≀ = accept ≀ in
    let m = receive^≀ in
    let n = receive^≀ in
    if pred(m) then
        select^≀ l1
        send^≀ f(m, n)
        send^≀ g(m, n, n)
    else
        select^≀ l2
        let o = receive^≀ in
        send^≀ f(n, m)
        send^≀ g(m, n, o)
```

***Session type classes*.** Type classes [5], [11] provide type-safe ad-hoc polymorphism. They allow the programmer to define a fixed set of functions over multiple datatypes, where each datatype has a bespoke implementation of each function in the set. We call these sets of functions type classes. They are usually implemented by *dictionary passing* [11]. That means that at compile time an additional argument (the dictionary) and suitable access to this argument are synthesised for all code depending on type classes. With implicit arguments we can make dictionary passing implicit, and type classes become a special case of implicit arguments. This is a common Scala idiom [8].

Implicit messages suggest a natural generalisation of type classes: pass access to dictionaries by implicit messages! We illustrate this with an example. In Haskell, `Show` is a type class that converts values to string representations. We generalise this to message-passing concurrency: instead of a conversion function, we have a conversion *server*. We show example implementations `intShow` and `boolShow` (with some details omitted). Additional function servers can be written against this code over types that define a `Show` type class server.

```
type Show = ?a. ?^≀⟨?a. !String.
    end⟩^a. !String. end

show :: ⟨Show⟩^a → end
show c =
    let         c = accept c           in
    let a     , c = receive c          in
    let aShow , c = implicit receive c in
    let         d = request aShow      in
    let         d = send a d           in
    let as    , d = receive d          in
    send as c

implicit boolShow :: ⟨?Bool. !String.
    end⟩^a → end
boolShow c =
    let     c = accept c  in
    let b , c = receive c in
    send (if b then "true" else "false") c

implicit intShow :: ⟨?Int. !String.
    end⟩^a → end
intShow = ...

showUser :: ⟨Show⟩^r → end
showUser ap =
    let    c = request ap in
    let    c = send 10 c  in
    let s, c = receive c  in
    printf(s) ;
    c
```

Clients communicating with the `show` server such as `showUser` do not need explicitly to send their show implementation, but send one implicitly.

It would be possible to make this example even more terse by eliminating repeated rebinding with implicit functions, however for clarity we show just one application at a time.

II. The language Im

This section presents our language Im of implicit message passing. Im is an extension of Gay and Vasconcelos's calculus of linear types for asynchronous sessions (LAST) [2]. (Familiarity to LAST will be essential for understanding the rest of the paper.) LAST is a λ-calculus with primitives for spawning threads that exchange messages. LAST was the first coherent integration of session types with λ-calculus and uses linear types at the λ-level to mediate between session types and functions. LAST is a suitable foundation for implicit message passing because its smooth integration of functions and processes enables us to provide both: implicit functions and implicit messages.

As the compiler synthesises the missing arguments at compile-time from type information, calculi for implicit arguments might be best understood not as programming languages, but as meta-programming systems that generate code in a base language L from input programs in L with implicits. Indeed, SI [7], an extension of System F, Scala's foundations for implicits, does not have a self-contained operational semantics, and is instead compiled to System F. We use the same approach, and translate Im to LAST.

Syntax. In the presentation of IM's syntax, let v range over values and e over expressions. We assume that x ranges over a countable set of term variables, c over a countable set of channel endpoints, n over $\mathbb{N} \cup \{\infty\}$, l over labels and I over finite subsets of $\mathbb{N}$. In order to make the presentation easily accessible, we highlight the extensions IM adds to LAST.

$$
\begin{array}{rcl}
v & ::= & \lambda x.e \mid (v, v) \mid \texttt{unit} \mid \texttt{fix} \mid \texttt{fork} \\
& \mid & \texttt{request}\ n \mid \texttt{accept}\ n \mid \texttt{send} \\
& \mid & \texttt{receive} \mid \texttt{implicit receive} \\
\\
e & ::= & v \mid e\ e \mid (e, e) \mid \texttt{let}\ x, x = e\ \texttt{in}\ e \\
& \mid & \texttt{select}\ l\ e \mid \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} \\
& \mid & \wr \mid \texttt{let}\ x, \wr = e\ \texttt{in}\ e
\end{array}
$$

Here `implicit receive` is the implicit analog of `receive`. Unlike `receive`, it is not matched by a corresponding `send`, but a corresponding `send` is inserted during translation, while `implicit receive` is translated into a normal `receive`. $\wr$ denotes a query to the implicit scope. $\wr$ is removed at translation time, and is replaced by a nondeterministically chosen name in the implicit scope. The construct `let x, ≀ = ...` allows us to add variables to the implicit scope, and as with the lone $\wr$, we also replace $\wr$ within `let` by a variable name during translation. Note that we often write `let ≀ = e in e'`. This is a convenience and can be thought of as syntactic sugar for `let _, ≀ = (_, e) in e'` where `_` is an unused variable or expression.

The parameter n following `accept` n and `request` n gives a *bound* for session communication. This will be explained in later sections. Note that we omit the bound parameter for brevity where not relevant.

An IM program is a configuration of expressions in parallel, running as separate threads and typed in a suitable environment. We now define configurations, ranged over by C.

$$
\begin{array}{rcl}
b & ::= & v \mid l \\
\\
C & ::= & C \parallel C \mid c \mapsto (c, n, \vec{b}) \mid (\nu cc)C \mid \langle e \rangle
\end{array}
$$

III. TYPES FOR IM

Just as SI is given meaning by type-guided translation to System F in [7], we give such a translation of IM into LAST. This section prepares the translation by extending LAST's typing system with types for implicit message passing and implicit functions. Types for IM are given by the following grammar. Here T ranges over types for the λ-calculus part of IM, S over session types, and B over buffer types.

$$
\begin{array}{rcl}
T & ::= & \texttt{Unit} \mid S \mid T \otimes T \mid T \to T \mid T \multimap T \\
& \mid & \langle S \rangle^{\mathrm{r}} \mid \langle S \rangle^{\mathrm{a}} \mid \langle S, S' \rangle \mid T \wr\!\!\to T \\
& \mid & T \wr\!\!\multimap T \\
\\
S & ::= & \texttt{end} \mid ?T.S \mid !T.S \mid \&\langle l_i : S_i \rangle_{i \in I} \\
& \mid & \oplus\langle l_i : S_i \rangle_{i \in I} \mid X \mid \mu X.S \mid ?^{\wr}T.S \\
& \mid & !^{\wr}T.S \\
\\
B & ::= & T \mid l
\end{array}
$$

The type $T \wr\!\!\to T$ is the type of implicit functions. It is written $? \to$ in [7] but we replace $?$ by $\wr$ to avoid confusion with the input session type $?T.S$. The type $T \wr\!\!\multimap T$ is the linear equivalent of $T \wr\!\!\to T$. As with [7], we do not have syntax for implicit abstraction and application - these are inferred during *implicit resolution* in Section IV.

The types $!^{\wr}T.S$ and $?^{\wr}T.S$ are the types of implicit message input and output respectively. They are the dual of one another as with explicit output and input. Implicit output types cannot be deduced from a process's syntax (since they are implicit) and must be inferred by inspecting the process that contains the corresponding implicit input. This happens during implicit resolution.

Buffer content types $\vec{B}$ are composed of vectors of entries B. Each entry is either a type T, representing the type of a value that is to be sent and stored in the buffer, or a label l representing the selection of such an option l by a process communicating using the buffer. Buffer content types $\vec{B}$ are assigned to buffers $\vec{b}$ such that for each v in $\vec{b}$ there exists a type T in the corresponding buffer content type $\vec{B}$ such that $v : T$. This notion is made precise in Section IV.

Given below are the type schemas for the constants k. They are the same as LAST's, and can be instantiated for any appropriate type.

$$
\begin{array}{lcl}
\texttt{fix} & : & (T \to T) \to T \\
\texttt{send} & : & T \to !T.S \multimap S \\
\texttt{send} & : & T \to !T.S \to S \quad \text{if } un(T) \\
\texttt{fork} & : & T \to \texttt{Unit} \quad \text{if } un(T) \\
\texttt{receive} & : & ?T.S \to T \otimes S \\
\texttt{request}\ n & : & \langle S \rangle^{\mathrm{r}} \to \overline{S} \quad \text{if } bound(\overline{S}) \leq n \\
\texttt{accept}\ n & : & \langle S \rangle^{\mathrm{a}} \to S \quad \text{if } bound(S) \leq n \\
\texttt{unit} & : & \texttt{Unit}
\end{array}
$$

Note that we omit a type schema for `implicit receive`. This is because it cannot be translated by the rule [T-CONST] in Figure 1, but needs a bespoke typing rule as unlike the other constants its translation is not identity.

We now give the session type duality function for our calculus. If a session type S and S' are *dual*, written

$\overline{S} = S'$, then a pair of terms of types S and S' can interact without communication errors. Such processes match in the sense that every action that one takes is matched by the other - if one outputs, the other inputs. If one offers a choice, the other makes a choice. We extend the duality function of LAST to include the two forms of implicit communication:

$$\begin{array}{rcl@{\qquad}rcl}
\overline{?T.S} & = & !T.\overline{S} & \overline{!T.S} & = & ?T.\overline{S} \\
\overline{?^\wr T.S} & = & !^\wr T.\overline{S} & \overline{!^\wr T.S} & = & ?^\wr T.\overline{S} \\
\overline{\mu X.S} & = & \mu X.\overline{S} & \overline{X} & = & X \\
\overline{\oplus\langle l_i : S_i\rangle_{i\in I}} & = & \&\langle l_i : \overline{S}_i\rangle_{i\in I} & \overline{\texttt{end}} & = & \texttt{end} \\
\overline{\&\langle l_i : S_i\rangle_{i\in I}} & = & \oplus\langle l_i : \overline{S}_i\rangle_{i\in I} & & &
\end{array}$$

We now define the subtyping relation coinductively by extension of the definitions for LAST.

Definition 1. A type T is *contractive* if it does not have subexpressions of the form $\mu X_1 ... \mu X_n . X_i$ where $0 < i \leq n$.

Let $\mathscr{S}$ denote the set of contractive, closed session types, and let $\mathscr{T}$ denote the set of types in which all session types are contractive and closed. We now define the function $F(\cdot)$ on binary relations over $\mathscr{T}$. We omit all cases not involving the new type constructs. The remaining clauses are formally identical with LAST's.

$$\begin{aligned}
F(R) = \ & ... \\
& \cup \{(?^\wr T.S, ?^\wr T'.S') | (T, T'), (S, S') \in R\} \\
& \cup \{(!^\wr T.S, !^\wr T'.S') | (T', T), (S, S') \in R\} \\
& \cup \{(T_1 \wr\!\!\to T_1', T_2 \wr\!\!\to T_2') | (T_2, T_1), (T_1', T_2') \in R\} \\
& \cup \{(T_1 \wr\!\!\multimap T_1', T_2 \wr\!\!\multimap T_2') | (T_2, T_1), (T_1', T_2') \in R\} \\
& \cup \{(T \wr\!\!\to T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\} \\
& \cup \{(T \to T', T \wr\!\!\to T') | T, T' \in \mathscr{T}\} \\
& \cup \{(T \multimap T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\}
\end{aligned}$$

Contractivity ensures that F is monotone. We write $T <: U$ if the pair (T, U) is in the greatest fixpoint of F. The last two lines in the definition of $F(\cdot)$ allow us to type sending *explicit* messages to *implicit* input.

The *matches* relation determines whether a given buffer type $\vec{B}$ agrees with a session type S. We write $\vec{B}\ mat\ S$ when the types in $\vec{B}$ match a prefix of those in S. We formalise this notion with the rules below:

$$\frac{\vec{B}\ \ mat\ \ S \quad U <: T}{U\vec{B}\ \ mat\ ?^\wr T.S}\ \text{M-OutI} \qquad \frac{\vec{B}\ \ mat\ \ S \quad U <: T}{U\vec{B}\ mat ?T.S}\ \text{M-Out}$$

$$\frac{-}{\epsilon\ \ mat\ \ S}\ \text{M-Empty} \qquad \frac{\vec{B}\ \ mat\ \ S}{l\vec{B}\ \ mat\ \ \&\langle ..., l : S, ...\rangle}\ \text{M-Case}$$

For some S and $\vec{B}$ such that $\vec{B}\ \ mat\ \ S$, $S/\vec{B}$ gives the session behaviours remaining as a *postfix* of S after performing those behaviours that correspond with $\vec{B}$. We define the postfix operator below:

$$\begin{array}{rcl@{\qquad}rcl}
S/\epsilon & = & S & ?T.S/U\vec{B} & = & S/\vec{B} \\
?^\wr T.S/U\vec{B} & = & S/\vec{B} & \&\langle ..., l : S, ...\rangle/l\vec{B} & = & S/\vec{B}
\end{array}$$

Next, we define $bound(S)$, which gives the *bound* of a session type, an upper bound on the runtime size of the buffer required to hold the values received on a channel with session type S. We start with the auxiliary operator $\mathsf{bds} \in (\mathscr{S} \to \mathbb{N}^\infty) \to \mathscr{S} \to \mathbb{N}^\infty$.

$$\mathsf{bds}(f)(S) = \begin{cases}
1 + f(S') & S \in \{?T.S', ?^\wr T.S'\} \\
1 + max\{f(S_i)\}_{i\in I} & S = \&\langle l_i : S_i\rangle_{i\in I} \\
f(S[^{\mu X.S'}/_X]) & S = \mu X.S' \\
0 & \text{otherwise}
\end{cases}$$

We now define the relation $S \mapsto S'$, which computes an *advanced* session type S' given a session type S.

$$\begin{array}{rcl@{\qquad}rcl}
?T.S & \mapsto & S & !T.S & \mapsto & S \\
?^\wr T.S & \mapsto & S & !^\wr T.S & \mapsto & S \\
\&\langle ..., l : S, ...\rangle & \mapsto & S & \oplus\langle ..., l : S, ...\rangle & \mapsto & S
\end{array}$$

$$\mu X.S \mapsto S' \text{ if } S\{\mu X.S/X\} \mapsto S'$$

We can now define $bound(S) = max\{\mu(S') | S \mapsto^* S'\}$ where μ is the least fixed point of bds.

IV. Translation from Im to LAST

This section presents implicit resolution, the type-directed translation of Im programs to LAST. We proceed in three steps, translation of expressions, translation of buffers and translation of configurations. Following [7], the translation is type-directed in that we give typing rules for Im, instrumented with translations to LAST. By forgetting the instrumentation, we obtain a typing system for Im.

Typing environments and implicit scope. Implicit resolution removes queries $\wr$ and inserts explicit functions and messages in place of implicit ones. This happens by choosing arguments from the implicit scope. We define the implicit scope thusly: The typing environment Γ is divided into two parts: the implicit and explicit scopes. That is to say, some of the bindings in Γ refer to implicit variables and some to explicit variables. In our typing rules we range over implicit variables with y and explicit variables with x. Variables enter the implicit scope in several ways: (1) when received as an implicit message; (2) when given as an argument to an implicit function; and (3) when bound by a `let` construct with $\wr$ on the left-hand side of the =.

Typing and translation of expressions. Typing judgements for expressions are of the form $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$.

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad T <: U}{\Gamma \vdash e : U \rightsquigarrow \widehat{e}}\ \text{T-Sub} \qquad \frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x_1 : T, x_2 : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x_1, x_2 = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x_1, x_2 = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2}\ \text{T-Split}$$

$$\frac{\Gamma_1 \vdash e_1 : T \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : U \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : T \otimes U \rightsquigarrow (\widehat{e}_1, \widehat{e}_2)}\ \text{T-Pair} \qquad \frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e} \quad un(\Gamma)}{\Gamma \vdash \lambda x.e : T \to U \rightsquigarrow \lambda x.\widehat{e}}\ \text{T-Abs} \qquad \frac{un(\Gamma) \quad k : T}{\Gamma \vdash k : T \rightsquigarrow k}\ \text{T-Const}$$

$$\frac{\Gamma_1 \vdash e_1 : T \multimap U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : T \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : U \rightsquigarrow \widehat{e}_1\ \widehat{e}_2}\ \text{T-App} \qquad \frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}}{\Gamma \vdash \lambda x.e : T \multimap U \rightsquigarrow \lambda x.\widehat{e}}\ \text{T-AbsL} \qquad \frac{un(\Gamma)}{\Gamma, \alpha : T \vdash \alpha : T \rightsquigarrow \alpha}\ \text{T-ID}$$

$$\frac{\Gamma_1 \vdash e : \&\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad \forall_{i \in I} (\Gamma_2 \vdash e_i : T_i \multimap T \rightsquigarrow \widehat{e}_i)}{\Gamma_1 + \Gamma_2 \vdash \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} : T \rightsquigarrow \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I}}\ \text{T-Case}$$

$$\frac{\Gamma \vdash e : \oplus\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad j \in I}{\Gamma \vdash \texttt{select}\ l_j\ e : T_j \rightsquigarrow \texttt{select}\ l_j\ \widehat{e}}\ \text{T-Select} \qquad \frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y\ \text{fresh}}{\Gamma \vdash e : T \multimap U \rightsquigarrow \lambda y.\widehat{e}}\ \text{T-AbsLI}$$

$$\frac{\Gamma_1 \vdash e : T \multimap U \rightsquigarrow \widehat{e} \quad \Gamma_2 \vdash \wr : T \rightsquigarrow y}{\Gamma_1 + \Gamma_2 \vdash e : U \rightsquigarrow \widehat{e}\ y}\ \text{T-AppI} \qquad \frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y\ \text{fresh} \quad un(\Gamma)}{\Gamma \vdash e : T \to U \rightsquigarrow \lambda y.\widehat{e}}\ \text{T-AbsI}$$

$$\frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2 \quad y\ \text{fresh}}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2}\ \text{T-SplitI} \qquad \frac{un(\Gamma)}{\Gamma, y : T \vdash \wr : T \rightsquigarrow y}\ \text{T-Query}$$

$$\frac{un(\Gamma)}{\Gamma \vdash \texttt{implicit receive} : ?^{\wr}T.S \to T \otimes S \rightsquigarrow \texttt{receive}}\ \text{T-InI} \qquad \frac{\Gamma_1 \vdash \wr : T \rightsquigarrow y \quad \Gamma_2 \vdash e : !^{\wr}T.S \rightsquigarrow \widehat{e}}{\Gamma_1 + \Gamma_2 \vdash e : S \rightsquigarrow \texttt{send}\ y\ \widehat{e}}\ \text{T-OutI}$$

Fig. 1. Type guided translation of expressions.

This can be read as "under assumptions Γ, the expression e has type T and is translated to the LAST expression $\widehat{e}$". Our typing and translation rules can be found in Figure 1. With the exception of the new syntactic forms of expressions, the translations are homomorphic, yielding rules similar in structure to those found in [2]. The rules for our new syntactic forms are more interesting. The rules [T-SplitI], [T-AppI], [T-AbsI] and [T-Query] follow a similar structure to those in [7]. Note that with [T-Query], the variable chosen to replace $\wr$ must satisfy linearity constraints, a restriction not present in [7]. [T-AbsLI] is a linear version of the rule for implicit functions and is effectively a combination of the rules [T-AbsI] and [T-AbsL]. The rule [T-InI] translates `implicit receive` into `receive` and otherwise behaves in the same way as [T-Const]. [T-OutI] translates implicit outputs by inserting a `send` action into the process. The argument for the `send` is a variable from the implicit scope, which we get from the first premise by translating $\wr$ with (a subset of) the input environment. This yields an implicit variable with the appropriate type whilst also satisfying any linearity constraints. Note that [T-OutI] is the only rule adding outputs directly.

Typing and translation of buffer contents. Typing judgements for buffers follow the same form as typing judgements for expressions. We write $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$. The translation of buffers can be found in Figure 2.

Typing and translation of configurations. Typing judgements for configurations (Figure 3) follow a slightly different form to those for buffer contents and expressions. We write $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$. This can be read as "under assumptions Γ, the configuration C yields buffer types Δ and is translated as $\widehat{C}$". We define buffer type maps Δ below in Definition 2. The rules [T-Thread], [T-Buffer] and [T-New] are as in [2], augmented with homomorphic translations. The rule [T-Par] is also similar to its equivalent rule in [2], but also contains two new premises. The first computes the buffer types in the configuration $C_1 \parallel C_2$, which are used in the second premise to perform *implicit resolution*. The judgements used in these premises are explained below.

We introduce a derivation of the form $S_1 \asymp S_2 \gg S_1', S_2'$. This judgement can be understood as saying that session types S_1 and S_2 are compatible if we rewrite them as S_1' and S_2'. The rule [C-Sub] captures the LAST notion of compatibility. In this case no augmentation of the compared types is required. The rule [C-Imp] accounts for implicit behaviour. Intuitively this rule says that if two session types are compatible except that one has an implicit input unmatched by the other, we can augment the other with a corresponding implicit output and judge them compatible. The [C-Rev] rule captures the symmetry enjoyed by the compatibility relation of LAST.

$$\frac{\overline{S}_1 <: S_2}{S_1 \asymp S_2 \gg S_1, S_2}\ \text{C-Sub} \qquad \frac{S_1 \asymp S_2 \gg S_1', S_2'}{S_2 \asymp S_1 \gg S_2', S_1'}\ \text{C-Rev}$$

$$\frac{S_1 \asymp S_2 \gg S_1', S_2'}{?^{\wr}T.S_1 \asymp S_2 \gg ?^{\wr}T.S_1, !^{\wr}T.S_2}\ \text{C-Imp}$$

Definition 2. *Buffer types* are triples of the form $(d, n, \vec{B})$. We let Δ range over partial finite maps from channel names to *buffer types* in C. $\Delta + \Delta'$ means that the domains of Δ and Δ' are disjoint.

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma \vdash l\vec{b} : l\vec{B} \rightsquigarrow \widehat{l\vec{b}}}\ \text{T-SeqL} \qquad \frac{un(\Gamma)}{\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon}\ \text{T-Empty}$$

$$\frac{\Gamma_1 \vdash v : T \rightsquigarrow \widehat{v} \quad \Gamma_2 \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B} \rightsquigarrow \widehat{v\vec{b}}}\ \text{T-SeqV}$$

Fig. 2. Type guided translation of buffers contents.

DEFINITION 3. We define a partial operation of addition on environments:

$$\Gamma + x : T = \begin{cases} \Gamma, x : T & x \notin \mathsf{dom}(\Gamma) \\ \Gamma & \Gamma(x) = T, \mathsf{un}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We extend this to $\Gamma + \Gamma'$ inductively from the base case.

DEFINITION 4. We say that Γ and Δ *resolve* to Γ', written $\Gamma, \Delta\, resolve\, \Gamma'$ provided there are environments Γ_p and Γ_s with *disjoint* domains such that:

- $\Gamma' = \Gamma_p + \Gamma_s$.
- Γ_p has the property that $\forall c, d \in dom(\Gamma) \cap dom(\Delta)$: assuming that
  - $\Delta(c) = (d, n, \vec{B})$.
  - $\Delta(d) = (c, n', \vec{B'})$.

  then $\Gamma(c)/\vec{B} \asymp \Gamma(d)/\vec{B'} \gg S, S'$ implies that $\Gamma_p(c) = S$ and $\Gamma_p(d) = S'$.
- Γ_s is Γ restricted to $\mathsf{dom}(\Gamma) \setminus \mathsf{dom}(\Gamma_p)$.

The *buf* relation $\Gamma \vdash C\ buf\ \Delta$ computes the buffer types present in a configuration C. In order to determine where output actions must be inserted during implicit resolution, we first determine those channels over which session communication occurs. The *buf* relation determines this by inspecting the structure of the configuration C and collecting the names of channels used for session communication, and their associated buffer types, into the map Δ. The rule [B-NEW] removes from the map Δ those channels that are restricted by a (νcd) construct. [B-BUFFER] extracts a channel name and associated buffer type using the environment Γ. [B-PAR] combines results across parallel composition. [B-THREAD] is an empty base case. The *buf* rules can be found in Figure 4.

Sources of nondeterminism. There are two sources of nondeterminism in implicit resolution. The first is in the selection of the implicit variable chosen by the rule [T-QUERY]. We do not specify which variable in the implicit scope should replace a ≀. A possible way to resolve this is to use nesting. Such a solution would select the innermost implicit variable of the appropriate type as the translation for ≀. The Scala compiler uses a more complex version of this strategy, augmented with other selection criteria [7].

$$\frac{\Gamma \vdash C\ buf\ \Delta}{\Gamma \setminus cd \vdash (\nu cd)C\ buf\ \Delta \setminus cd}\ \text{B-New} \qquad \frac{-}{\Gamma \vdash \langle e \rangle\ buf\ \emptyset}\ \text{B-Thread}$$

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \_}{\Gamma \vdash c \mapsto (d, n, \vec{b})\ buf\ c : (d, n, \vec{B})}\ \text{B-Buffer}$$

$$\frac{\Gamma_1 \vdash C_1\ buf\ \Delta_1 \quad \Gamma_2 \vdash C_2\ buf\ \Delta_2}{\Gamma_1 + \Gamma_2 \vdash C_1 \parallel C_2\ buf\ \Delta_1 + \Delta_2}\ \text{B-Par}$$

Fig. 4. The *buf* relation. $\Gamma \setminus cd$ is Γ restricted to variables $\notin \{c, d\}$.

The second source of nondeterminism results from the insertion of output actions when resolving implicit messages. When a pair of composed processes are resolved, we do not specify which is resolved first. As a result, adjacent implicit inputs can be resolved in multiple ways. Consider the processes:

```
⟨ let p =
    let ≀ = ...
    let c = accept x in
    let n, ≀ = implicit receive in c
in p ⟩ ‖ ⟨ let q =
    let ≀ = ...
    let d = request x in
    let n, ≀ = implicit receive in d
in q ⟩
```

Implicit resolution should insert two output actions here, one in `p` and the other in `q`. If we resolve `p` before `q`, we obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    send y c
    let n, c = receive in c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
    let n, d = receive in d
    send y d
in q ⟩
```

We could also resolve `q` first and obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    let n, c = receive in c
    send y c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
    send y d
    let n, d = receive in d
in q ⟩
```

As with nondeterminism caused by resolution of ≀, an implementation could use a simple heuristic such as to resolve the accepting partner before the requesting partner. We leave this question for future work.

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad un(T)}{\Gamma \vdash \langle e \rangle \triangleright \emptyset \rightsquigarrow \langle \widehat{e} \rangle}\ \text{T-Thread} \qquad \frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}} \quad |\vec{b}| \leq n}{\Gamma \vdash c \mapsto (d, n, \vec{b}) \triangleright c : (d, n, \vec{B}) \rightsquigarrow c \mapsto (d, n, \widehat{\vec{b}})}\ \text{T-Buffer}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_1 \parallel C_2 \; buf \; \Delta \quad \Gamma, \Delta \; resolve \; \Gamma' \quad \Gamma' = \Gamma'_1 + \Gamma'_2 \quad \Gamma'_1 \vdash C_1 \triangleright \Delta_1 \rightsquigarrow \widehat{C}_1 \quad \Gamma'_2 \vdash C_2 \triangleright \Delta_2 \rightsquigarrow \widehat{C}_2 \quad \Delta' = \Delta_1 + \Delta_2 \\ \forall c \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \Rightarrow (\vec{B} \;\; mat \;\; \Gamma'(c) \; and \; bound(\Gamma'(c)) \leq n)) \\ \forall c, d \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \; and \; \Delta'(d) = (c, n', \vec{B}') \Rightarrow \Gamma'(c)/\vec{B} <: \Gamma'(d)/\vec{B}')\end{array}}{\Gamma \vdash C_1 \parallel C_2 \triangleright \Delta' \rightsquigarrow \widehat{C}_1 \parallel \widehat{C}_2}\ \text{T-Par}$$

$$\frac{\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \triangleright \Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2) \rightsquigarrow \widehat{C}}{\Gamma \vdash (\nu c_1 c_2) C \triangleright \Delta \rightsquigarrow (\nu c_1 c_2) \widehat{C}}\ \text{T-New}$$

Fig. 3. Type guided translation of configurations.

V. Runtime safety of Im

We prove safety of Im's translation into LAST: we show that if we can derive $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$ in Im, then $\widehat{C}$ can be typed suitably in LAST. In order to make this precise, we define a function $(\cdot)^*$, that translates Im's types to standard LAST types. This translation simply erases occurrences of $\wr$, yielding non-implicit analogues of implicit types.

Definition 5 (Translation of types)**.**

$$\begin{array}{rclcrcl}
(T \wr\!\!\to T')^* & = & T^* \to T'^* & \qquad & \texttt{Unit}^* & = & \texttt{Unit} \\
(T \wr\!\!\multimap T')^* & = & T^* \multimap T'^* & & \texttt{end}^* & = & \texttt{end} \\
(T \to T')^* & = & T^* \to T'^* & & (?T.S)^* & = & ?T^*.S^* \\
(T \multimap T')^* & = & T^* \multimap T'^* & & (!T.S)^* & = & !T^*.S^* \\
\&\langle l_i : S_i \rangle_{i \in I}^* & = & \&\langle l_i : S_i^* \rangle_{i \in I} & & (\langle S \rangle^{\mathtt{r}})^* & = & \langle S^* \rangle^{\mathtt{r}} \\
\oplus\langle l_i : S_i \rangle_{i \in I}^* & = & \oplus\langle l_i : S_i^* \rangle_{i \in I} & & (\langle S \rangle^{\mathtt{a}})^* & = & \langle S^* \rangle^{\mathtt{a}} \\
\langle S, S' \rangle^* & = & \langle S^*, S'^* \rangle & & (?^{\wr} T.S)^* & = & ?T^*.S^* \\
(T \otimes T')^* & = & T^* \otimes T'^* & & (!^{\wr} T.S)^* & = & !T^*.S^* \\
(\mu X.S)^* & = & \mu X.S^* & & X^* & = & X
\end{array}$$

We extend the definition of $(\cdot)^*$ to buffer types:

Definition 6 (Translation of buffer types and environments)**.**

$$\begin{array}{rclcrcl}
\epsilon^* & = & \epsilon & \qquad & (T\vec{B})^* & = & T^*\vec{B}^* \\
(l\vec{B})^* & = & l\vec{B}^* & & (c, n, \vec{B})^* & = & (c, n, \vec{B}^*)
\end{array}$$

This is lifted pointwise to environments (i.e. $(\Gamma, x : T)^* = \Gamma^*, x : T^*$).

We call a configuration *fully buffered* if whenever it contains $c \mapsto (c', n, \vec{b})$ then it also contains $c' \mapsto (c, n', \vec{b}')$. We recall the following theorem from [2], defining and proving LAST's runtime safety.

Theorem. Let $\Gamma \vdash_{\text{LAST}} C \triangleright \Delta$ be a fully buffered LAST configuration, and assume that $C \longrightarrow^* C'$. If C'' is a blocked thread in C', then one if the following applies: (1) C'' is $\langle v \rangle$ or $\langle \texttt{send}\ v \rangle$ or $\langle \texttt{request}\ n\ v \rangle$ or $\langle \texttt{accept}\ n\ v \rangle$; (2) C'' is $\langle E[\texttt{receive}\ c] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$; (3) C'' is $\langle E[\texttt{case}\ c\ \texttt{of}\ \{l_i : e_i\}_{i \in I}] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$.

We state our main result.

Theorem 1 (Runtime safety of Im)**.** If $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$ is a fully buffered Im configuration, then $\Gamma^* \vdash_{\text{LAST}} \widehat{C} \triangleright \Delta^*$ is a runtime-safe LAST configuration.

Proof. By induction on $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$. □

VI. Further work

Implicits are useful in sequential programming not just for type classes, but also e.g. for generic programming [9]. With implicits at hand, it is possible to investigate generic programming in message passing systems. Such technology transfer from the sequential to concurrent computation would be aided by a better understanding of the relationship between implicit functions and implicit messages. Finally, we conjecture that implicits are not tied to binary sessions, and can be adapted to multi-party session types [4].

References

[1] R. Atkey. Parameterised notions of computation. *JFP*, 19(3-4):335–376, 2009.

[2] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):1950, 2010.

[3] K. Honda. Types for dyadic interaction. In *Proc. CONCUR*, 1993.

[4] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Proc. POPL*, pages 273–284, 2008.

[5] S. Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proc. ESOP*, pages 131–144, 1988.

[6] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proc. POPL*, 2000.

[7] M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, and S. Stucki. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. POPL*, 2018.

[8] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proc. OOPSLA*, pages 341–360, 2010.

[9] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: A new foundation for generic programming. In *Proc. PLDI*, pages 35–44, 2012.

[10] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *Proc. PARLE*, 1994.

[11] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proc. POPL*, pages 60–76, 1989.