Science of Computer Programming 75 (2010) 1165–1191

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Type inference and strong static type checking for Promela

Alastair F. Donaldson [a,*], Simon J. Gay [b]

[a] Oxford University Computing Laboratory, Oxford, OX1 3QD, UK

[b] Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK

ARTICLE INFO

Article history:
Received 24 September 2009
Received in revised form 27 May 2010
Accepted 28 May 2010
Available online 18 June 2010

Keywords:
Promela
Spin
Type checking
Type inference
Model checking

ABSTRACT

The Spin model checker and its specification language Promela have been used extensively in industry and academia to check the logical properties of distributed algorithms and protocols. Model checking with Spin involves reasoning about a system via an abstract Promela specification, thus the technique depends critically on the soundness of this specification. Promela includes a rich set of data types including first-class channels, but the language syntax restricts the declaration of channel types so that it is not generally possible to deduce the complete type of a channel directly from its declaration. We present the design and implementation of Etch, an enhanced type checker for Promela, which uses constraint-based type inference to perform strong type checking of Promela specifications, allowing static detection of errors that Spin would not detect until simulation/verification time, or that Spin may miss completely. We discuss theoretical and practical problems associated with designing a type system and type checker for an existing language, and formalise our approach using a Promela-like calculus. To handle subtyping between base types, we present an extension to a standard unification algorithm to solve a system of equality and subtyping constraints, based on *bounded* substitutions.

1. Introduction

The Spin model checker [16] and its specification language Promela have been used extensively in industry and academia to check the logical properties of distributed algorithms and protocols (see *e.g.* [4,12,17,32]). Model checking with Spin involves reasoning about a system via an abstract Promela specification, thus the technique depends critically on the soundness of this specification. Promela includes a rich set of data types, including first-class communication channels which can be transmitted along other channels in the style of the pi calculus [31]. However, although type information for program variables is declared and checked, the language allows information about the types of channels (*i.e.* the allowed types of messages) to be only partially specified, and there are situations in which a complete specification is not even permitted.

Channel declarations in Promela fall into the following three categories.

1. Declarations that specify complete type information, for example:

```
chan a = [0] of {int,int}
```

   which declares a channel whose buffer has size 0 (a rendezvous channel) and which can carry messages consisting of pairs of integers.
2. Declarations that specify no message type information, for example:

```
chan b
```

* Corresponding author.
E-mail addresses: alastair.donaldson@comlab.ox.ac.uk (A.F. Donaldson), simon@dcs.gla.ac.uk (S.J. Gay).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.05.010

```
chan a = [0] of {chan};

proctype P() {
  chan b = [0] of int;
  int i;
  ... a!b; b!i; ...
}

proctype Q() {
  chan c;
  byte j;
  ... a?c; c?j; ...
}
```

Fig. 1. An erroneous Promela specification.

3. Declarations of channels that carry channels, for example:

```
chan c = [0] of {chan}
```

In this case the programmer does *not* have the option of replacing the innermost `chan` with a more precise type, *e.g.* the following declaration is not allowed:

```
chan d = [0] of {chan{int,int}}
```

Declarations in category (1) are unproblematic and allow the SPIN type checker to check the types of messages sent and received.[1] But declarations in categories (2) and (3) limit the scope of the type checker. For example, consider the Promela specification in Fig. 1. Here `a` is a global channel, `b` is a channel sent from `P` to `Q`, and `c` is the channel reference into which `Q` receives `b`. At runtime, `c` becomes instantiated to `b`, and the send command `b!i` synchronizes with the receive command `c?j`. The result is that the value of `i`, which is an int, is put into the variable `j`, which is a byte. We regard this as a type error, since the range of values belonging to the byte type is smaller than those for the int type. However, the type checker built into SPIN is not able to detect this error, because the type of channel `a` is not fully specified and therefore both `P` and `Q` seem to be using it correctly.

Situations such as the example above frequently, although not always, signify errors in the Promela specification; to put it another way, they indicate errors in the modelling of the system which is to be analyzed. This can lead to errors during model checking, perhaps after a long period of checking; alternatively, it can lead to unexpected behaviour of the model and therefore to misleading results from model checking. In the example above, assigning an integer value into a byte variable might be allowed by SPIN, but the user might not know what the runtime behaviour (*e.g.* truncation or saturation) will be.

The essential problem is that Promela allows mobile channels, but does not use an appropriate type system for mobile channels. We have addressed this problem by designing ETCH (**E**nhanced **T**ype **CH**ecker), an enhanced type checking tool for Promela. ETCH facilitates the analysis of Promela specifications with mobile channels using an appropriate type system, but does *not* require changes to the syntax of the Promela language. ETCH is based on the following type-theoretic technology:

Fully specified channel types, as found in type systems for the pi calculus [31]. For example, the type:

chan{int, chan{bool, chan{int}}}

could be associated with a channel which can only be declared in Promela as:

```
chan a = [0] of {int,chan}
```

Constraint-based type inference, as found in functional languages such as ML [22] and Haskell [25]. This allows fully specified channel types to be used internally without requiring the programmer to declare them explicitly. In particular, this means that we do not need to modify the syntax of Promela.

Recursive types, again as found in functional languages and in several type systems for the pi calculus. Recursive types naturally arise while solving constraints among channel types. For example, the declaration `chan a` together with the send command `a!a` requires channel `a` to have a recursive type defined by the equation $X = \mathsf{chan}\{X\}$. The programmer never needs to write a recursive type, but they can appear in error messages, and are presented in as palatable a form as possible.

Subtyping, which accounts for the relationships between the various numeric types supported by Promela. The subtyping relation is defined by:

$\mathsf{bit} <: \mathsf{byte} <: \mathsf{short} <: \mathsf{int} \qquad \mathsf{bit} <: \mathsf{bool}.$

The constraint-based type inference system generates constraints involving subtyping relationships as well as equalities. In general this complicates constraint solving, but it turns out that in Promela there is always just enough explicit type information to enable us to obtain a straightforward algorithm.

[1] Even so, the type checking carried out by the SPIN tool remains minimal: the tool follows a weak type system where channels and integers are partially interchangeable.

Error	Detected statically	Detected during simulation	Detected during verification	Leads to erroneous verification result
1	No	No	No	Yes, 190 states without error *vs.* 182 states with error
2	No	Indirectly	Yes	No, error detected dynamically
3	No	No	No	No, but depends on Spin representing mtype and chan identically
4	No	Yes	No	Yes, 116607 states without error *vs.* 119061 states with error

Fig. 2. Summary of the ability of Spin to detect Errors 1– 4 statically, during simulation, or during verification. In some cases, the error goes undetected and leads to an erroneous verification result in the sense that the number of reachable states explored by Spin is affected by the presence of the error. Our enhanced type checker, Etch, is able to detect all these errors statically.

Etch is able to detect many situations which can reasonably be regarded as type errors but which are not detected by Spin. However, because Etch is implemented as a standalone tool, Spin users are free to view its error messages as warnings and ignore them if they choose.

1.1. Contribution and structure of the paper

The main contribution of this paper is the design and implementation of an enhanced type checker for Promela, which can statically detect errors that Spin would not detect until simulation/verification time, or that Spin may miss completely. The tool, Etch, is publicly available[2] to the Promela/Spin community, to aid the construction of robust specifications for formal verification. The paper also provides a case study in applying, and adapting in a non-trivial way, well-understood techniques from type theory to an existing language whose syntax we did not want to change. We believe that by providing a concrete exposition of type-theoretic techniques for a practical, imperative language, our case study will be a useful reference for practitioners who are not experts in type theory.

We further motivate the need for enhanced type checking of Promela via a discussion of example Promela specifications, which also provides an overview of the language (Section 2). After this, we make the following contributions:

- We introduce Promela-calculus, a simple language based on Promela but with complete type information. We present a type system for Promela-calculus, and establish that well-typed Promela-calculus specifications are free of communication errors (Section 3).
- We consider a variant of Promela-calculus in which type information is incompletely specified, as is the case for Promela. We present a constraint-based type checking method that guarantees that specifications with soluble sets of constraints are free from communication errors. The approach is based on standard techniques for type reconstruction, extended to handle subtyping between base types via *bounded substitutions* (Section 4).
- Although our results are presented in terms of Promela-calculus, our implementation Etch is for the full Promela language. We present notes of some interesting practical issues, including the use of most a general unifier for inferring further properties of channel usage, and the way our implementation deals with recursive types (Section 5).

We conclude (Section 7) after a discussion of related work (Section 6). An operational semantics for Promela-calculus, as well as proofs of novel results, appear in the Appendix.

2. Examples of errors detected by Etch

To motivate the need for applying type inference to Promela, and to illustrate the kind of type errors which Etch detects, we present two example Promela specifications from the literature, into each of which we inject two errors. In each case, Spin does not detect the error before simulation or verification, and in some cases does not catch the error at all. This is summarised in Fig. 2. Etch, on the other hand, uses the techniques described later in the paper to detect these errors statically, before simulation or verification of the model, making it easier to eliminate them. We also discuss a scenario where static type checking can be restrictive. Results in this section were obtained using Spin version 5.2.4.

2.1. Client–server specification

We first consider a generic client–server specification adapted from [16, Chapter 15], where a detailed description of the protocol is provided. The Promela code for this specification is given in Fig. 3.

The specification introduces an enumerated type, `mtype`, to represent messages in the protocol, and declares two global channel variables: `server` and `null`. Both channels are rendezvous (having length zero), and accept pairs of messages consisting of an `mtype` and a channel. Three process types are then declared, via the `proctype` construct. The *Client* proctype

[2] Etch can be downloaded from: http://www.allydonaldson.co.uk/etch, or from the Spin website: http://spinroot.com.

```
1  mtype = {request,deny,hold,grant,return};
2  chan server = [0] of {mtype,chan};
3  chan null = [0] of {mtype,chan}
4
5  proctype Agent(chan listen, talk) {
6    do :: talk!hold,listen
7       :: talk!deny,listen -> break
8       :: talk!grant,listen ->
9  wait:   listen?return,null; break
10   od;
11   server!return,listen
12 }
13
14 active[2] proctype Client() {
15   chan me = [0] of {mtype,chan};
16   chan agent;
17 end:
18   do :: timeout -> server!request,me;
19         do :: me?hold,agent
20            :: me?deny,agent -> break
21            :: me?grant,agent -> agent!return,null; break
22         od
23   od
24 }
25
26 active proctype Server() {
27   chan agents[2] = [0] of {mtype,chan};
28   chan pool = [2] of {chan};
29   chan client, agent; byte i;
30   do :: i < 2 -> pool!agents[i]; i++
31      :: else -> break
32   od;
33 end:
34   do :: server?request,client ->
35         if :: empty(pool) -> client!deny,null
36            :: nempty(pool) -> pool?agent;
37               run Agent(agent,client)
38         fi
39      :: server?return,agent -> pool!agent
40   od
41 }
```

Fig. 3. Client–server specification.

is prefixed with `active [2]` to indicate that two instances of this proctype should be live when simulation or verification of the specification begins; the *Server* proctype is prefixed simply with `active`, indicating that one instance of this proctype should be live. The *Agent* proctype has no such prefix: this indicates that no instances of this proctype are live initially, instead instances are spawned via the `run` statement at line 37. This non-blocking statement instantiates an asynchronous *Agent* with the given parameters. Statements in Promela are separated either by `;` or `->`, which can be used interchangeably. However, it is common practice for `->` to follow a boolean guard, *e.g.* `timeout` at line 18 (`timeout` is a built-in boolean variable which has value *true* if and only if every other process is blocked). Looping behaviour is specified using the `do`..`od` construct; this construct selects between a series of options, each prefixed `::`, non-deterministically choosing between options for which the first statement is executable. The `break` keyword causes a jump to the end of the innermost `do`..`od` construct. The specification makes frequent use of channels, using the send (`!`) and receive (`?`) operators, as well as built-in functions to test whether or not a channel is empty.

The specification is interesting to study as it exhibits dynamic channel passing — the *Server* process holds a pool of channels which are used to connect *Agent* and *Client* processes, and *Agent* processes use the *server* channel to return their input channels to the pool. Also, the specification involves implicit recursive types.

Fig. 4(a) shows the output generated by Etch for this example. Etch reports that the specification is well typed, and displays the complete type for each variable. All channels with the exception of `pool` are found to have the recursive type satisfying the equation $X = \mathsf{chan}\{X, \mathsf{mtype}\}$. We write this type as $\mathsf{rec}\, X\, .\, \mathsf{chan}\{X, \mathsf{mtype}\}$. The `pool` channel accepts messages which are channels of the above type, thus the type of `pool` is $\mathsf{chan}\{\mathsf{rec}\, X\, .\, \mathsf{chan}\{X, \mathsf{mtype}\}\}$.

We consider two changes to the client–server specification which could conceivably arise due to programmer errors.

Error 1. *Statement* `talk!hold,listen` *at* 6 *of Fig.* 3 *is replaced with* `talk!listen,hold`.

From the specification, we can see that *Agent* processes are instantiated only by the *Server* process at line 37. The *talk* parameter of an *Agent* corresponds to the *client* variable of the *Server* process, which is defined at line 34 by a receive on the *server* channel. The corresponding sender for this statement is a *Client* process, which sends the channel *me* at line 18. Channel *me* accepts messages of the form {mtype, chan}. Thus the *talk* parameter of an *Agent* accepts messages of the form {mtype, chan}. The modification introduces an error since an attempt is made to send a message of the form {chan, mtype} on the channel *talk*.

```
Client
------
  agent  : rec X.chan{mtype,X}
  me     : rec X.chan{mtype,X}

Server
------
  i      : byte
  agent  : rec X.chan{mtype,X}
  agents : array(size 2) of
           rec X.chan{mtype,X}
  client : rec X.chan{mtype,X}
  pool   : chan{rec X.chan
                {mtype,X}}

Agent
-----
  talk   : rec X.chan{mtype,X}
  listen : rec X.chan{mtype,X}

Globals
-------
  null   : rec X.chan{mtype,X}
  server : rec X.chan{mtype,X}
```

(a) Reconstructed channel types for client–server specification.

```
User
----

  messchan  : rec X.chan{X,bit}
  partnerid : byte
  messbit   : bit
  dev       : mtype
  self      : rec X.chan{X,bit}
  state     : mtype
  selfid    : byte



Globals
-------

  one       : rec X.chan{X,bit}
  null      : rec X.chan{X,bit}
  partner   : array(size 3) of
              rec X.chan{X,bit}
  two       : rec X.chan{X,bit}
  zero      : rec X.chan{X,bit}
```

(b) Reconstructed channel types for telephone specification.

Fig. 4. Examples of reconstructed types output by ETCH for well-typed specifications.

As indicated in Fig. 2, this error is not detected by SPIN **at all**. The reachable state-space associated with the specification differs depending on whether or not the error is present, thus Error 1 leads to an erroneous verification result.

In contrast, ETCH detects this error statically, producing the following error message:

```
Error at line 6: "mtype" is not compatible with type "rec X.chan{X,mtype}".
```

Error 2. *Statement* `talk!grant,listen` *at* 8 *of Fig. 3 is replaced with* `talk!grant`.

By an argument similar to that for Error 1, this modification causes an error since only a single field has been sent on the channel *talk*. This error is not detected statically by SPIN. During simulation, SPIN does not report an error when this statement is executed. The message is received by a *Client* process via the statement `me?grant(agent)` at line 21. However, the received channel *agent* is uninitialised because no channel was actually sent by the *Agent* process. The *Client* process then attempts to execute `agent!return(null)`. This causes an error during simulation as *agent* is not initialised. Thus SPIN indirectly catches the error during simulation. During verification, SPIN immediately halts with an error when the statement of Error 2 is executed, thus this error does not lead to an erroneous verification result. The behaviour for SPIN with respect to Error 2 is summarised in Fig. 2.

ETCH detects this error statically, producing the following message:

```
Error at line 8: arguments of lengths 1 and 2 have been used with the same channel.
```

2.2. Telephone specification

The second specification we consider is shown, in part, in Fig. 5. This specification models a telephone network, and is adapted from a specification presented in full in [4], in which the telephone system is augmented with a selection of features, and SPIN is used for feature interaction analysis. The specification illustrates a more explicit use of recursive channel types than for the client–server example of Section 2.1, shows the capability of ETCH to detect *subtyping* errors via type reconstruction, and shows how ETCH can be used to trap specification errors which alter the state-space of the associated model, but which are not detected by SPIN at all, even during verification.

This specification uses several Promela features which were not illustrated by Fig. 3. Buffered channels are specified via a non-zero length — the channels `null`, `zero`, `one` and `two` all have capacity 1. At line 4, an array of three channels is declared. The specification uses the `if..fi` construct for conditional selection (this is similar to the `do..od` construct discussed in Section 2.1, except that it does not cause looping behaviour), and includes labelled statements which may be the targets for `goto` instructions, in the style of the C language. At line 16, arguments to the receive operator are enclosed in angle brackets: this indicates that the receive should be non-destructive, copying values from the channel into the supplied arguments but not removing a message from the channel. Rather than specifying initial processes via the `active` keyword, this specification declares an `init` process, which is live by default when verification or simulation begins. The `init` process initialises the

```
1   mtype = { on, off, st_idle };
2   chan null = [1] of {chan, bit}; chan zero = [1] of {chan, bit};
3   chan one  = [1] of {chan, bit}; chan two  = [1] of {chan, bit};
4   chan partner[3]
5
6   proctype User (byte selfid; chan self) {
7     chan messchan = null; bit messbit = 0;
8     mtype state = on, dev = on;
9     byte partnerid = 6;
10
11  O_T_Null:
12    if :: empty(self) -> state = on;
13          dev = off;
14          self!self,0;
15          goto Auth_Orig_Att
16       :: full(self) -> self?<partner[selfid],messbit>;
17          if :: empty(partner[selfid]) -> self?messchan,messbit;
                                                                    ...
84    fi;
85
86    if :: empty(partner[selfid]) ->
87          partner[selfid]!self,0;
88          self?messchan,messbit;
89          self!partner[selfid],0;
                                                                    ...
286   fi
287 }
288
289 init {
290   atomic {
291     partner[0]=null; partner[1]=null; partner[2]=null;
292     run User(0,zero); run User(1,one); run User(2,two);
293 } }
```

Fig. 5. Part of a telephone specification, adapted from [4].

`partner` array, and instantiates three *User* processes. These statements are enclosed in an **`atomic`** block, to specify that they should be executed indivisibly.

Each *User* process in the specification takes a channel parameter, *self*, which accepts data in pairs consisting of a channel reference (the channel of a *User* process to whom the given *User* is connected), and a bit representing the connection status of the call. Statement `self!self,0` at line 14 of Fig. 5 shows that it permissible for the channel *self* to hold a reference to itself. This indicates that the corresponding *User* process is engaged, but not connected [4]. Fig. 4(b) shows the complete type information for this specification as reconstructed by Etch. Notice the appearance of recursive types of a similar form to those detected by Etch in Fig. 4(a).

Again, we consider the introduction of potential programmer errors to the specification.

Error 3. *Declaration* **`chan`** `two = [1]` **`of`** `{`**`chan, bit`**`}` *at* 3 *is replaced with* **`chan`** `two = [1]` **`of`** `{`**`mtype, bit`**`}`.

This modification intuitively introduces a type error: the channel *two* is passed as parameter *self* to a *User* process at line 292, and at line 14 the statement `self!self,0` indicates that the first field of a message on *self* should be a channel, not an mtype expression (which is the type of the first field of *two* in Error 3).

In practice, as indicated in Fig. 2, Spin does not flag this up as an error, statically or dynamically. Spin uses a byte to represent the value of both chan and mtype variables, so assigning between chan and mtype variables does not cause any loss in precision. For this reason and because the channel *self* is consistently used as if its first field were a channel, the model associated with the specification of Fig. 5 is the same whether or not Error 3 is introduced.

We view Error 3 as a genuine error for three reasons. Firstly, using the type name mtype where chan would be more sensible makes the specification harder to understand. Secondly, relying on the fact that Spin treats mtype and chan variables identically provides little guarantee that a specification will behave as expected with future version of Spin (or with other tools that use Promela as an input language (*e.g.* p2b [3]). Lastly, the Xspin user interface provides a feature which allows the values of variables to be displayed during simulation. This feature replaces the numeric values associated (internally) with mtype variables by the symbolic names provided by the user via the **`mtype`** `= {...}` declarator. This means that in the presence of Error 3 messages on channel *two* will be displayed using mtype names rather than channel names for their first field, making simulation confusing.

Etch detects this error statically and displays the following message:

```
Error at line 17: "chan{mtype,bit}" is not compatible with type "mtype".
```

Error 4. *Statement* `self!partner[selfid],0` *at* 89 *is replaced with* `self!partner[selfid],9`.

In this final erroneous modification to the telephone specification, we assume that the user has mistyped the value 0 as 9. The Spin tool includes the *bit* data type for convenience, but does not complain if a value outside {0, 1} is used in a *bit*

```
mtype = { ho_cmd, ... };

proctype CC(chan f, ...) {  /* communication controller */
  chan  m_new;
  do :: l?m_new -> f!ho_cmd; f!m_new;
        ...
}

proctype BS(chan f, ...) {  /* base station */
  chan v; ...
  do :: f?ho_cmd -> f?v;    /* handover command */
        ...
}
```

Fig. 6. A snippet of the `mobile1` specification, provided with the SPIN distribution, which illustrates a limitation of static type checking with ETCH. Proctypes `CC` and `BS` communicate both mtype and chan messages on their channel parameter, `f`, thus ETCH reports a type error. Nevertheless, the full specification is free of communication errors.

context. SPIN uses an approach similar to that of C with respect to boolean expressions, and treats the value 0 as usual, and any other value as 1. This means that the statement `self!partner[selfid],9` means the same thing as `self!partner[selfid],1`, which is clearly different to the intended statement.

As with Error 1, this error leads to an erroneous verification result: SPIN does not detect the error statically or during verification, and the size of the reachable state-space associated with the telephone specification differs depending on whether or not Error 4 is present. During simulation, SPIN does report a truncation error when the statement of Error 4 is executed.

ETCH detects the error statically, with the following message:

```
Error at line 89: type "byte" occurs in a context where it is required to be a subtype of "bit".
```

Errors 1 and 4 are arguably the most serious of the errors we have discussed, since they go undetected by SPIN but lead to a semantic difference in the associated models, illustrated by the change in state-space size shown in Fig. 2. In the presence of such errors, the user might try to verify properties of the erroneous specification, accepting that certain properties are true, when in fact the properties hold *vacuously*: they have not been verified over the specification intended by the user.

2.3. The price of static type checking

Before we present the techniques on which our enhanced type checker is based, it is fair to point out that while static type checking can quickly identify genuine errors, a static type system always leads to the rejection of some correct programs. In our context, "correct" means Promela specifications that, when executed, do not exhibit communication errors.

An important example of this is the `mobile1` specification provided with the SPIN distribution. This specification models a cell-phone hand-off strategy in a mobile network, and is translated from a pi calculus description [24]. A small fragment of the specification is shown in Fig. 6. Instances of proctypes `CC` (communication controller) and `BS` (base station) are spawned, with the same channel passed as parameter `f`. The `CC` process receives a channel into local variable `m_new`, then sends two messages on channel `f`: a hand-off command (represented by mtype value `ho_cmd`), followed by the channel stored in `m_new`. Correspondingly, the `BS` process uses `f` to receive the hand-off command, followed by the channel.

ETCH rejects this specification, since channel `f` in both proctypes is used as if it had both type chan{mtype} *and* chan{chan{...}}. However, the specification executes without communication errors since, as discussed for Error 3 in Section 2.2, SPIN uses the same internal representation for mtype and chan values. Furthermore, the `mobile1` specification is designed such that whenever a chan value is sent on a channel, the corresponding receive is into a chan variable, and similarly for mtype values.

The specification can be re-written to be accepted by ETCH, by representing messages on `f` as (mtype, chan) pairs. Then each communication using an mtype is replaced with a communication using the same mtype and a null channel, and each communication using a chan replaced with a communication using a null mtype and the same chan. The advantage of the representation is that the distinction between mtype and chan messages is made explicit, and the specification is deemed well typed. The price is that this representation requires a slightly larger state vector, since the width of channel variables is increased. We note that this is the only example provided with the SPIN distribution that is beyond the scope of ETCH.

The scenario where the type of message to be communicated via a channel depends on the status of a communications protocol can often be captured using session types [18], which have been successfully applied to practical concurrent programming [7,10]. An interesting area for future work is a formulation of session types suitable for Promela, to allow static checking of efficient protocols where channel fields are used with multiple message types in a structured way.

3. A Type system with complete type information

Since no standard formal semantics for Promela is available, we present our type checking algorithm, and prove its correctness, with respect to *Promela-calculus* (*PC*), a small language based on the pi calculus. *PC* captures the features

Values	v	::=	`true` | `false` | *numeric literal*	
Types	T	::=	bool | bit | byte | short | int	
		|	chan{T, . . . , T}	channel (with product type for message fields)
		|	X	type variable
		|	μX.T	recursive type
Expressions	e	::=	v	literal
		|	x	variable
		|	x = e	assignment
		|	e → e : e	conditional
		|	e==e	equality
		|	x ! e, . . . , e	send
		|	x ? x, . . . , x	receive
		|	e; e	sequence
		|	T x; e	declaration
		|	`run` P(e, . . . , e)	process instantiation
Definitions	D	::=	`proctype` P(T x; . . . ; T x) { e }	
Specification	S	::=	D . . . D e	

Fig. 7. Top-level syntax for Promela-calculus with complete type information (PC_{full}).

```
proctype P(chan{chan{int}} link, int request)
{
  chan{int} response;
  int expected;

  expected = 10000;
  link?response;
  expected==request ->
     response!20000 : response!0
}

chan{chan{int}} a;
chan{int} b;
int result;
run P(a, 10000);
a!b;
b?result;
result
```

Fig. 8. Example Promela-calculus specification.

of Promela in which we are primarily interested: channel-based communication with support for first-class channels, recursive channel types, and subtyping between base types, and omits orthogonal language features which would add to the complexity of our proofs without providing further insight into the problems we are interested in solving. Note that our implementation, Etch, supports the Promela language in full.

In this section we present a version of *PC* in which channel types are *fully* specified. We introduce a standard type system for this language, and state properties of interest: a type preservation theorem, and a result showing that execution of a well-typed specification cannot result in communication errors.

3.1. Syntax

The top-level syntax for *PC* is given in Fig. 7, and an example specification presented in Fig. 8.

A *PC* specification consists of a series of proctype declarations, followed by an expression, which can be thought of as a *main* process. A simple expression is a literal value (v), variable reference (x), or receive operation (x ? x, . . . , x). Forms of compound expression are assignment (x = e), conditional (e → e : e),[3] comparison (e==e), send (x ! e, . . . , e),

[3] In C-like programming languages, the notation e ? e : e is used for conditional expressions. Promela uses → in place of ? to avoid parsing conflicts, due to the use of ? as the receive operator.

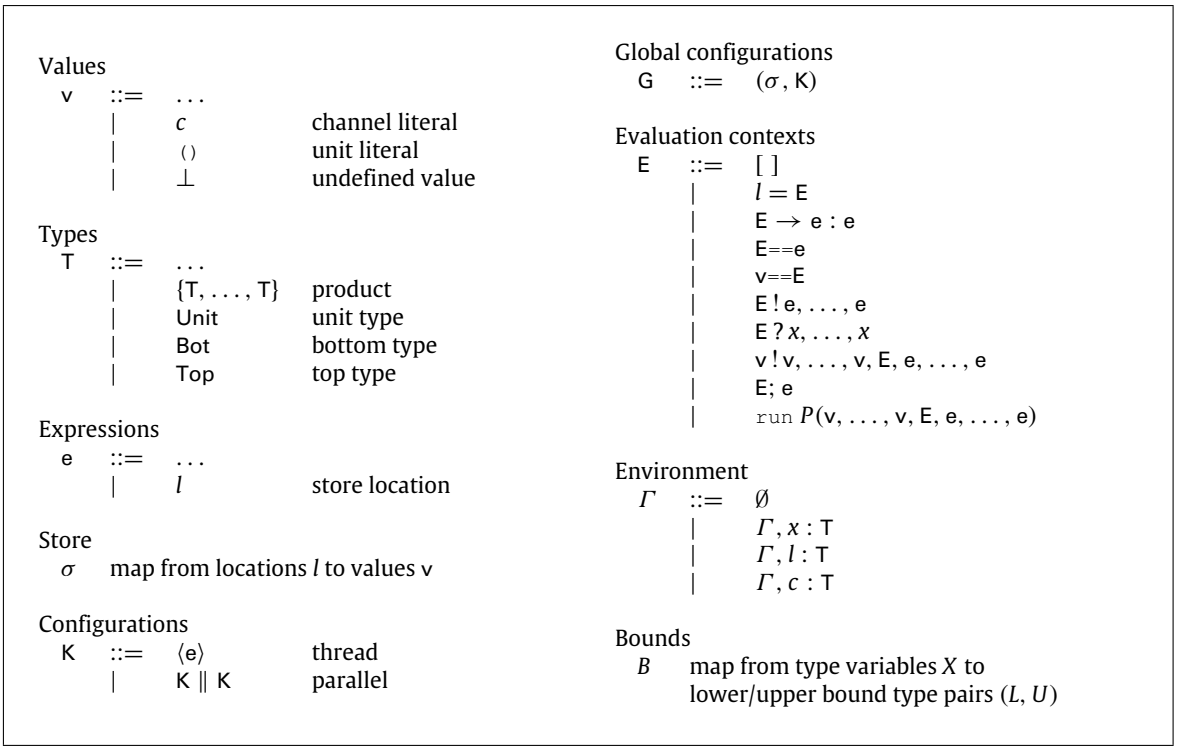

Fig. 9. Internal syntax for Promela-calculus.

process instantiation (run $P(e, \ldots, e)$) and sequence (e; e). A proctype is a named, parameterised process definition. Many instances of a proctype can be created to run in parallel via run expressions.

We use $\mathcal{A}$ to denote the set of *PC* base types: $\mathcal{A} = \{\text{bit, byte, short, int, bool}\}$. The values of type bool are `true` and `false` as usual, and the values of bit are 0 and 1. We do not specify values for the other types, we merely require that the literals inhabiting the numeric types satisfy the following containment relation: bit $\subseteq$ byte $\subseteq$ short $\subseteq$ int. Channel types are constructed by specifying an ordered list of field types. We follow the Promela convention of using curly braces to specify the list of field types. The language also provides recursive types, to cater for the sort of channel-based communication discussed in Section 2. We require that recursive types are contractive and types in declarations are closed.

A simple example *PC* specification is shown in Fig. 8. The *main* process launches an instance of proctype *P* with a channel and an integer request. The instantiated proctype uses the provided channel to receive a channel through which to send a response. If the integer request is the expected value of 10 000 then a value of 20 000 is sent back via the response channel, otherwise the value 0 is sent. The *main* process waits for a response from the instance of *P*, and terminates with the value received.

3.2. Semantics and internal syntax

An operational semantics for Promela-calculus is provided as Appendix A. Fig. 9 presents necessary syntax additional to that of Fig. 7 used to describe the progress of a Promela-calculus specification as it is evaluated using the operational semantics. The semantics track variables via store, σ which maps locations to values; store locations l are another form of expression. We use an *undefined* literal value $\perp$ for variables which have been defined but not assigned, and a set of channel literals c to represent channel identifiers to which channel variables may refer. A *configuration* K is a parallel composition of threads, where each thread has the form $\langle e \rangle$. The expression e defines the behaviour of a given thread, and the parallel composition of threads e_1 and e_2 is denoted $\langle e_1 \rangle \parallel \langle e_2 \rangle$. Combining a configuration K with a store σ yields a *global* configuration (σ, K), which corresponds to the notion of parallel threads sharing memory. Syntax for *evaluation contexts* [36] allow a concise presentation of the operational semantics.

Fig. 9 also presents syntax used by the type systems of Sections 3.3 and 4. The Unit type, with domain $\{()\}$ caters for expressions which do not yield numeric or boolean values. In Section 4, types Bot and Top are used to place bounds on type variables, and standalone product types are used to represent message field types in isolation. Syntax is also provided for environments, which bind variable names, store locations and channel literals to types, and for bounds, which associated upper and lower bound types with type variables and are used extensively in Section 4.

$\boxed{B; \Gamma \vdash \mathsf{e} : \mathsf{T}}$

$$\frac{b \text{ is a boolean literal}}{B; \Gamma \vdash b : \mathsf{bool}} \text{ (T-Bool-Lit)} \qquad \frac{n \text{ is a numeric literal} \quad \mathsf{T}_n \text{ is its type}}{B; \Gamma \vdash n : \mathsf{T}_n} \text{ (T-Num-Lit)}$$

$$\frac{}{B; \Gamma \vdash () : \mathsf{Unit}} \text{ (T-Unit-Lit)} \qquad \frac{}{B; \Gamma \vdash \bot : \mathsf{T}} \text{ (T-Undefined)}$$

$$\frac{x : \mathsf{T} \in \Gamma}{B; \Gamma \vdash x : \mathsf{T}} \text{ (T-Var)} \qquad \frac{l : \mathsf{T} \in \Gamma}{B; \Gamma \vdash l : \mathsf{T}} \text{ (T-Location)}$$

$$\frac{c : \mathsf{T} \in \Gamma}{B; \Gamma \vdash c : \mathsf{T}} \text{ (T-Channel)} \qquad \frac{\mathsf{e} : \mathsf{T} \in \Gamma \qquad B; \Gamma \vdash \mathsf{e}' : \mathsf{T}}{B; \Gamma \vdash \mathsf{e} = \mathsf{e}' : \mathsf{T}} \text{ (T-Assign)}$$

$$\frac{x : \mathsf{chan}\{\mathsf{T}_1, \ldots, \mathsf{T}_n\} \in \Gamma \quad B; \Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \ \ldots \ B; \Gamma \vdash \mathsf{e}_n : \mathsf{T}_n}{B; \Gamma \vdash x\,!\,\mathsf{e}_1, \ldots, \mathsf{e}_n : \mathsf{Unit}} \text{ (T-Send)}$$

$$\frac{x : \mathsf{chan}\{\mathsf{T}_1, \ldots, \mathsf{T}_n\} \in \Gamma \quad x_1 : \mathsf{U}_1 \in \Gamma \ \ldots \ x_n : \mathsf{U}_n \in \Gamma \quad B; \mathsf{T}_1 <: \mathsf{U}_1 \ \ldots \ B; \mathsf{T}_n <: \mathsf{U}_n}{B; \Gamma \vdash x\,?\,x_1, \ldots, x_n : \mathsf{Unit}} \text{ (T-Receive)}$$

$$\frac{B; \Gamma \vdash \mathsf{e}_1 : \mathsf{T} \qquad B; \Gamma \vdash \mathsf{e}_2 : \mathsf{T}}{B; \Gamma \vdash \mathsf{e}_1 == \mathsf{e}_2 : \mathsf{bool}} \text{ (T-Eq)} \qquad \frac{B; \Gamma \vdash \mathsf{e}_1 : \mathsf{T} \qquad B; \Gamma \vdash \mathsf{e}_2 : \mathsf{U}}{B; \Gamma \vdash \mathsf{e}_1; \mathsf{e}_2 : \mathsf{U}} \text{ (T-Seq)}$$

$$\frac{B; \Gamma \vdash \mathsf{e}_1 : \mathsf{bool} \quad B; \Gamma \vdash \mathsf{e}_2 : \mathsf{T} \qquad B; \Gamma \vdash \mathsf{e}_3 : \mathsf{T}}{B; \Gamma \vdash \mathsf{e}_1 \rightarrow \mathsf{e}_2 : \mathsf{e}_3 : \mathsf{T}} \text{ (T-Cond)} \qquad \frac{x \notin dom(\Gamma) \quad B; \Gamma, x : \mathsf{T} \vdash \mathsf{e} : \mathsf{U}}{B; \Gamma \vdash \mathsf{T}\,x\,;\ \mathsf{e} : \mathsf{U}} \text{ (T-Decl)}$$

$$\frac{\texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \} \text{ is defined} \quad B; \Gamma \vdash \texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \}\ \texttt{OK} \quad B; \Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \ \ldots \ B; \Gamma \vdash \mathsf{e}_n : \mathsf{T}_n}{B; \Gamma \vdash \texttt{run}\ P(\mathsf{e}_1, \ldots, \mathsf{e}_n) : \mathsf{Unit}} \text{ (T-Run)} \qquad \frac{B; \Gamma \vdash \mathsf{e} : \mathsf{T} \quad B \vdash \mathsf{T} <: \mathsf{U}}{B; \Gamma \vdash \mathsf{e} : \mathsf{U}} \text{ (T-Sub)}$$

$\boxed{B; \Gamma \vdash \mathsf{D}\ \texttt{OK}}$

$$\frac{B; \Gamma \vdash \mathsf{T}_1\ x_1;\ \ldots;\ \mathsf{T}_n\ x_n;\ \mathsf{e} : \mathsf{T}}{B; \Gamma \vdash \texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \}\ \texttt{OK}} \text{ (OK-Proctype)}$$

$\boxed{B \vdash \mathsf{T} <: \mathsf{U}}$

$$\frac{}{B \vdash \mathsf{bit} <: \mathsf{bool}} \text{ (T-Bit-Bool)} \qquad \frac{}{B \vdash \mathsf{bit} <: \mathsf{byte}} \text{ (T-Bit-Byte)}$$

$$\frac{}{B \vdash \mathsf{byte} <: \mathsf{short}} \text{ (S-Byte-Short)} \qquad \frac{}{B \vdash \mathsf{short} <: \mathsf{int}} \text{ (S-Short-Int)}$$

$$\frac{}{B \vdash \mathsf{T} <: \mathsf{T}} \text{ (S-Refl)} \qquad \frac{B \vdash \mathsf{T} <: \mathsf{U} \qquad B \vdash \mathsf{U} <: \mathsf{V}}{B \vdash \mathsf{T} <: \mathsf{V}} \text{ (S-Trans)}$$

$$\frac{}{B \vdash \mathsf{Bot} <: \mathsf{T}} \text{ (S-Bot)} \qquad \frac{}{B \vdash \mathsf{T} <: \mathsf{Top}} \text{ (S-Top)} \qquad \frac{B(X) = (\mathsf{L}, \mathsf{U})}{B \vdash \mathsf{L} <: X} \text{ (S-Lower)} \qquad \frac{B(X) = (\mathsf{L}, \mathsf{U})}{B \vdash X <: \mathsf{U}} \text{ (S-Upper)}$$

$\boxed{B; \Gamma \vdash \mathsf{K}\ \texttt{OK}}$

$$\frac{B; \Gamma \vdash \mathsf{e} : \mathsf{T}}{B; \Gamma \vdash \langle \mathsf{e} \rangle\ \texttt{OK}} \text{ (OK-Thread)} \qquad \frac{B; \Gamma \vdash \mathsf{K}\ \texttt{OK} \qquad B; \Gamma \vdash \mathsf{K}'\ \texttt{OK}}{B; \Gamma \vdash \mathsf{K} \parallel \mathsf{K}'\ \texttt{OK}} \text{ (OK-Parallel)}$$

$\boxed{B; \Gamma \vdash \sigma\ \texttt{OK}}$

$$\frac{\forall\, l \in dom(\sigma)\ .\ B; \Gamma \vdash \sigma(l) : \Gamma(l)}{B; \Gamma \vdash \sigma\ \texttt{OK}} \text{ (OK-Store)}$$

$\boxed{B; \Gamma \vdash \mathsf{G}\ \texttt{OK}}$

$$\frac{B; \Gamma \vdash \sigma\ \texttt{OK} \qquad B; \Gamma \vdash \mathsf{K}\ \texttt{OK}}{B; \Gamma \vdash (\sigma, \mathsf{K})\ \texttt{OK}} \text{ (OK-Global)}$$

$\boxed{B; \Gamma \vdash \mathsf{S} : \mathsf{T}}$

$$\frac{B; \Gamma \vdash \mathsf{D}_1\ \texttt{OK} \ \ldots \ B; \Gamma \vdash \mathsf{D}_n\ \texttt{OK} \qquad B; \Gamma \vdash \mathsf{e} : \mathsf{T}}{B; \Gamma \vdash \mathsf{D}_1\ \ldots\ \mathsf{D}_n\ \mathsf{e} : \mathsf{T}} \text{ (T-Spec)}$$

Fig. 10. Type system for Promela-calculus.

3.3. Type system

A type system for *PC* is presented in Fig. 10. Typing judgements are with respect to both an environment Γ (as standard) and a bounds function B, which assigns free type variables to lower and upper type bounds. This bounds function will be used extensively in Section 4 when we consider constraint-based typing, and until then can be ignored.

The T-Bool-Lit, T-Num-Lit and T-Unit-Lit rules give types to literal values in the obvious way. The undefined literal value $\perp$ may be assigned to any store location, so we need to be able to give this literal value *any* type: this requirement is satisfied by rule T-Undefined. Rules T-Var, T-Location, T-Channel allow us to look up the type of a variable, store location or channel literal based on the type recorded for the entity in the environment.

The typing rules T-Assign, T-Cond, T-Eq, T-Send and T-Receive formally specify requirements of *PC* operations, *e.g.* that the number of argument expressions sent on a channel must match the arity of the channel, and that the type of argument i must match the type of the ith element in the product type associated with the channel. Our type system assumes that recursive channel types are unfolded sufficiently so that their outermost type constructor is chan. Sequences of expressions are handled by the rule T-Seq, which states that the type of a typable sequence is the type of the last expression in the sequence. Declarations are checked by rule T-Decl, which requires that the name of the declared variable is not already in use and the remainder of the expression is typable under the environment extended with the new variable. Type Checking of proctype instantiations via `run` statements is achieved via the T-Run. This rule works by using the OK-Proctype rule to type check the associated proctype in the current environment, thus a single proctype declaration is type checked separately each time it is instantiated by a `run` statement.

The subtyping relation is presented by rules with conclusions of the form $\mathsf{T} <: \mathsf{U}$. Rules S-Lower and S-Upper allow subtyping information to be obtained for type variables based on the bounds function B. Rules S-Bot and S-Top mean that Bot and Top can be used as default lower and upper bounds for type variables.

The remainder of the typing rules are used to check that threads and parallel compositions of threads are well typed, and that the store associated with a thread is well formed. Rule T-Spec allows us to assign a type to a specification based on the type of the *main* expression for the specification, if this expression is typable.

By standard techniques [26] we can prove that typability is preserved by reductions in the operational semantics:

Theorem 3.1. *If* $\emptyset; \Gamma \vdash (\sigma, \mathsf{K})\ \mathit{OK}$ *and* $(\sigma, \mathsf{K}) \rightarrow (\sigma', \mathsf{K}')$ *then there exists* $\Gamma' \supseteq \Gamma$ *such that* $\emptyset; \Gamma' \vdash (\sigma', \mathsf{K}')\ \mathit{OK}$.

The syntax of types in Fig. 7 and the restrictions on well-formed types discussed in Section 3.1 mean that channel types are fully specified in a *PC* specification. As a result, the typing rules of Fig. 10 are sufficient to enable checking of the correctness of channel operations directly. In particular, the rules ensure that the correct number of arguments are supplied to send and receive operations, and that these arguments have appropriate types. This leads to the following theorem, also proved using standard methods:

Theorem 3.2. *If* $\emptyset; \Gamma \vdash (\sigma, \langle \mathsf{E}[c\,?\,l_1, \ldots, l_m] \rangle \parallel \langle \mathsf{E}'[c\,!\,v_1, \ldots, v_n] \rangle \parallel \mathsf{K})\ \mathit{OK}$ *then* $m = n$ *and* $\emptyset; \Gamma \vdash v_i : \Gamma(l_i)$ *for all* $i \in \{1, \ldots, n\}$.

Theorems 3.1 and 3.2 together show that execution of a well-typed *PC* specification cannot lead to a communication error.

4. Checking under-specified channel types using constraint-based type inference

We now introduce a variant of Promela-calculus where, as in Promela, channel types are only *partially* specified. We show that full type checking is still possible, despite the lack of channel type information, using type reconstruction techniques. We adapt a standard constraint-based type inference methods, following the presentation given in [26, Chapter 22]. Our approach consists of two parts:

- A *constraint typing relation*, defined by a set of syntax-directed typing rules. These rules give rise to a type checking algorithm that never fails: given a Promela-calculus specification, the algorithm generates a set of equality and subtyping constraints over type variables.
- A *unification algorithm*, which processes the set of constraints generated by the constraint typing relation and attempts to find a *solution*—a substitution of type variables for concrete types which essentially "fills in" the missing types from the Promela-calculus specification such that it is typable in the standard type system of Section 3. If the constraints are unsatisfiable, the unification algorithm fails, meaning that the Promela-calculus specification is not typable in the system of Section 3, no matter how the missing type information is filled in.

The standard approach of [26, Chapter 22] applies to the simply typed lambda calculus with integers and booleans, *without* subtyping. In general, combining subtyping with type reconstruction is complicated. However, in the scenario where we only have subtyping between base types we show that type reconstruction is possible.

We shall henceforth refer to Promela-calculus with complete type information (see Section 3) as PC_{full}, and Promela-calculus with partial type information (defined below) as $PC_{partial}$, unless the variant of the calculus referred to is clear from the context.

The grammar for $PC_{partial}$ is the same as for PC_{full} (see Fig. 7) except for the syntax of types, which is presented for $PC_{partial}$ in Fig. 11. The $PC_{partial}$ type syntax does not allow explicit specification of recursive types, and does not allow field types for channels to be specified beyond one level of nesting. A free type variable is provided for the fields of each channel declaration when left unspecified. This syntax reflects the situation in Promela, where channel types cannot be specified beyond one level of nesting.

$$
\begin{array}{lll}
\mathsf{T} & ::= & \mathsf{T}_{inner} \mid \mathsf{chan}\{\mathsf{T}_{inner}, \ldots, \mathsf{T}_{inner}\} \\
\mathsf{T}_{inner} & ::= & \mathsf{bool} \mid \mathsf{bit} \mid \mathsf{byte} \mid \mathsf{short} \mid \mathsf{int} \\
 & & \mid \mathsf{chan}\ X
\end{array}
$$

Fig. 11. Syntax for types in $PC_{partial}$. The rest of the top-level syntax is as for PC_{full} presented in Fig. 7.

A PC_{full} specification can be turned into a $PC_{partial}$ specification by unfolding all recursive types until any recursive type constructors appear deeper than one level of nesting, then replacing every channel field specifier deeper than one level of nesting with a distinct type variable. For example, unfolding the PC_{full} type $\mu X.\mathsf{chan}\{\mathsf{int}, X\}$ gives $\mathsf{chan}\{\mathsf{int}, \mathsf{chan}\{\mathsf{int}, \mu X.\mathsf{chan}\{\mathsf{int}, X\}\}\}$. Replacing the inner field specifier with a fresh type variable, Y say, yields the $PC_{partial}$ type $\mathsf{chan}\{\mathsf{int}, \mathsf{chan}\ Y\}$.

When analysing and reasoning about $PC_{partial}$ specifications we use the full language of types presented in Fig. 7, augmented with the internal type syntax of Fig. 9.

For ease of comparison, throughout this section we annotate a definition or result with (cf. [26, n]) to indicate that it is analogous to definition or result n in [26, Chapter 22].

4.1. Bounded substitutions

Recall from Section 3.1 that $\mathcal{A}$ denotes the set of Promela-calculus base types. We use $\mathcal{A}^*$ to denote $\mathcal{A} \cup \{\mathsf{Bot}, \mathsf{Top}\}$. Let $\mathcal{T}$ denote the set of all Promela-calculus types and $\mathcal{TV}$ the set of all type variable names.

Definition 4.1 (*cf. [26, 22.1.1]*)**.** A *bounded type substitution* (or just bounded substitution) is a pair (B, σ) where B is a partial function from $\mathcal{TV}$ to $\mathcal{A}^* \times \mathcal{A}^*$ and σ is a partial function from $\mathcal{TV}$ to $\mathcal{T}$, satisfying:

1. If $B(X) = (L_X, U_X)$ then $\emptyset \vdash L_X <: U_X$, $L_X \neq \mathsf{Top}$ and $U_X \neq \mathsf{Bot}$
2. $dom(B) \cap dom(\sigma) = \emptyset$
3. $FV(range(\sigma)) \subseteq dom(B)$.

For a type variable X and a bounded substitution (B, σ) there are four possible cases:

1. $\sigma(X)$ is defined, say $\sigma(X) = \mathsf{T}$, and T contains no type variables. In this case X is substituted for a concrete type.
2. $\sigma(X)$ is defined, say $\sigma(X) = \mathsf{T}$, and T contains type variables $X_1, \ldots, X_n$ such that $B(X_i)$ is defined for each X_i. In this case, X is substituted for an abstract type, parameterised by pairs of allowable bounds.
3. $\sigma(X)$ is undefined, but $B(X)$ is defined as (L_X, U_X). In this case, X can be any type $T \in \mathcal{A}$ lying between L_X and U_X. (Condition 1 of Definition 4.1 ensures that we do not have $L_X = U_X = \mathsf{Bot}$ or $L_X = U_X = \mathsf{Top}$.)
4. $\sigma(X)$ and $B(X)$ are both undefined. In this case X can be any type.

We can obtain a concrete substitution for every type variable by composing σ with a new substitution γ which assigns each type variable X satisfying case 3 above to a type $T \in \mathcal{A}$ between L_X and U_X, and assigns each type variable satisfying case 4 above to any concrete type. In practice, we consider bounded substitutions where B supplies bounds for every type variable of interest, so that case 4 does not apply.

4.2. Constraint-based typing

Definition 4.2 (*cf. [26, 22.3.1]*)**.** A *constraint* set $\mathcal{C}$ is a set of equations $\{T_i \bowtie_i U_i \ ^{i \in 1..n}\}$, where $\bowtie_i \in \{=, <:\}$ $(1 \leq i \leq n)$. A bounded substitution (B, σ) is said to *unify* (or *satisfy*) equation $T = U$ if the substitution instances $\sigma(T)$ and $\sigma(U)$ are identical (up to equivalence of recursive types). The substitution (B, σ) is said to unify (or satisfy) equation $T <: U$ if $B \vdash \sigma(T) <: \sigma(U)$. We say that (B, σ) is a *unifier* for $\mathcal{C}$ if it unifies every equation in $\mathcal{C}$, in which case we also say that (B, σ) satisfies $\mathcal{C}$.

4.2.1. Constraint typing relation

The *constraint typing relation* is defined by the rules of Fig. 12. The notation for presenting constraints is adapted from [26], with the addition of subtyping constraints. We read $\Gamma \vdash \mathsf{e} : \mathsf{T} \mid_\chi \mathcal{C}$ as "expression e has type T under assumptions Γ whenever constraints $\mathcal{C}$ are satisfied" [26]. The other forms of rule are interpreted similarly. The subscripts χ are used to track type variables appearing in sub-derivations, to ensure that distinct sub-derivations do not share type variable names.

The rules of Fig. 12 are syntax directed. Our type checker employs these rules as an algorithm that computes a type T, constraint set $\mathcal{C}$ and set of type variables χ given an environment Γ and $PC_{partial}$ specification $\mathsf{D}_1 \ldots \mathsf{D}_n\ \mathsf{e}$. We say that the constraint set $\mathcal{C}$ is *generated* by the constraint typing relation. In Section 4.3, we present a unification algorithm which computes the most general unifier for a constraint set, if any unifier exists.

$\boxed{\Gamma \vdash \mathsf{e} : \mathsf{T} \mid_{\chi} \mathcal{C}}$ $\mathcal{C}$ is a set of equality and subtyping constraints, χ is a set of type variables

$$\frac{b \text{ is a boolean literal}}{\Gamma \vdash b : \mathsf{bool} \mid_{\emptyset} \{\}} \text{ (CT-Bool-Lit)} \qquad \frac{\begin{array}{c}n \text{ is a numeric literal} \\ \mathsf{T}_n \text{ is its type}\end{array}}{B; \Gamma \vdash n : \mathsf{T}_n \mid_{\emptyset} \{\}} \text{ (CT-Num-Lit)}$$

$$\frac{\begin{array}{c}x \notin dom(\Gamma) \qquad \text{type variables in } \mathsf{T} \text{ all different} \\ \chi_1 = \text{type variables in } \mathsf{T} \qquad \Gamma, x : \mathsf{T} \vdash \mathsf{e} : \mathsf{U} \mid_{\chi_2} \mathcal{C}\end{array}}{\Gamma \vdash \mathsf{T}\ x;\ \mathsf{e} : \mathsf{U} \mid_{\chi_1 \cup \chi_2} \mathcal{C}} \text{ (CT-Decl)} \qquad \frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T} \mid_{\emptyset} \{\}} \text{ (CT-Var)}$$

$$\frac{x : \mathsf{T} \in \Gamma \qquad \Gamma \vdash e : \mathsf{U} \mid_{\chi} \mathcal{C} \qquad \mathcal{C}' = \mathcal{C} \cup \{\mathsf{U} <: \mathsf{T}\}}{\Gamma \vdash x = \mathsf{e} : \mathsf{T} \mid_{\chi} \mathcal{C}'} \text{ (CT-Assign)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_2} \mathcal{C}_2 \qquad \Gamma \vdash \mathsf{e}_3 : \mathsf{T}_3 \mid_{\chi_3} \mathcal{C}_3 \\ \mathsf{T}_2, \mathsf{T}_3 \in \mathcal{A}^* \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}\}\end{array}}{\Gamma \vdash \mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3 : \mathsf{T}_2 \vee \mathsf{T}_3 \mid_{\chi_1 \cup \chi_2 \cup \chi_3} \mathcal{C}'} \text{ (CT-Cond-Base)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_2} \mathcal{C}_2 \qquad \Gamma \vdash \mathsf{e}_3 : \mathsf{T}_3 \mid_{\chi_3} \mathcal{C}_3 \\ \mathsf{T}_2 \notin \mathcal{A}^* \text{ or } \mathsf{T}_3 \notin \mathcal{A}^* \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}, \mathsf{T}_2 = \mathsf{T}_3\}\end{array}}{\Gamma \vdash \mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3 : \mathsf{T}_2 \mid_{\chi_1 \cup \chi_2 \cup \chi_3} \mathcal{C}'} \text{ (CT-Cond-Compound)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_2} \mathcal{C}_2 \\ X \notin \chi_1, \chi_2, \mathsf{T}_1, \mathsf{T}_2, \mathcal{C}_1, \mathcal{C}_2, \Gamma, \mathsf{e}_1, \mathsf{e}_2 \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathsf{T}_1 <: X, \mathsf{T}_2 <: X\}\end{array}}{\Gamma \vdash \mathsf{e}_1 == \mathsf{e}_2 : \mathsf{bool} \mid_{\chi_1 \cup \chi_2 \cup \{X\}} \mathcal{C}'} \text{ (CT-Eq)}$$

$$\frac{\begin{array}{c}x : \mathsf{T} \in \Gamma \qquad \Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1 \ \ldots\ \Gamma \vdash \mathsf{e}_n : \mathsf{T}_n \mid_{\chi_n} \mathcal{C}_n \\ X_j \neq X_i \notin \chi_1, \ldots, \chi_n, \mathsf{T}_1, \ldots, \mathsf{T}_n, \mathcal{C}_1, \ldots, \mathcal{C}_n, \Gamma, e_1, \ldots, e_n\ (1 \leq i \neq j \leq n) \\ \mathcal{C}' = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{T} = \mathsf{chan}\{X_1, \ldots, X_n\}, \mathsf{T}_1 <: X_1, \ldots, \mathsf{T}_n <: X_n\}\end{array}}{\Gamma \vdash x\,!\,\mathsf{e}_1, \ldots, \mathsf{e}_n : \mathsf{Unit} \mid_{\chi_1 \cup \cdots \cup \chi_n \cup \{X_1, \ldots, X_n\}} \mathcal{C}'} \text{ (CT-Send)}$$

$$\frac{\begin{array}{c}x : \mathsf{T} \in \Gamma \qquad x_1 : \mathsf{T}_1 \in \Gamma \ \ldots\ x_n : \mathsf{T}_n \in \Gamma \\ X_j \neq X_i \notin \Gamma\ (1 \leq i \neq j \leq n) \\ \mathcal{C} = \{\mathsf{T} = \mathsf{chan}\{X_1, \ldots, X_n\}, X_1 <: \mathsf{T}_1, \ldots, X_n <: \mathsf{T}_n\}\end{array}}{\Gamma \vdash x\,?\,x_1, \ldots, x_n : \mathsf{Unit} \mid_{\{X_1, \ldots, X_n\}} \mathcal{C}} \text{ (CT-Receive)}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_2} \mathcal{C}_2 \qquad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash \mathsf{e}_1\ ;\ \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_1 \cup \chi_2} \mathcal{C}'} \text{ (CT-Seq)}$$

$$\frac{\begin{array}{c}\texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \} \text{ is defined} \\ \Gamma \vdash \texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \}\ \texttt{OK} \mid_{\chi} \mathcal{C} \\ \Gamma \vdash \mathsf{e}_1 : \mathsf{U}_1 \mid_{\chi_1} \mathcal{C}_1 \ \ldots\ \Gamma \vdash \mathsf{e}_n : \mathsf{U}_n \mid_{\chi_n} \mathcal{C}_n \qquad \mathcal{C}' = \mathcal{C} \cup \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{U}_1 <: \mathsf{T}_1, \ldots, \mathsf{U}_n <: \mathsf{T}_n\}\end{array}}{\Gamma \vdash \texttt{run}\ P(\mathsf{e}_1, \ldots, \mathsf{e}_n) : \mathsf{Unit} \mid_{\chi \cup \chi_1 \cup \cdots \cup \chi_n} \mathcal{C}'} \text{ (CT-Run)}$$

$\boxed{\Gamma \vdash \mathsf{D}\ \texttt{OK} \mid_{\chi} \mathcal{C}}$

$$\frac{\Gamma \vdash \mathsf{T}_1\ x_1;\ \ldots;\ \mathsf{T}_n\ x_n;\ \mathsf{e} : \mathsf{T} \mid_{\chi} \mathcal{C}}{\Gamma \vdash \texttt{proctype}\ P(\mathsf{T}_1\ x_1, \ldots, \mathsf{T}_n\ x_n)\ \{\ \mathsf{e}\ \}\ \texttt{OK} \mid_{\chi} \mathcal{C}} \text{ (C-OK-Proctype)}$$

$\boxed{\Gamma \vdash \mathsf{S} : \mathsf{T} \mid_{\chi} \mathcal{C}}$

$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\}.(\Gamma \vdash \mathsf{D}_i\ \texttt{OK} \mid_{\chi_i} \mathcal{C}_i) \\ \chi_1, \ldots, \chi_n \text{ pairwise disjoint} \qquad \Gamma \vdash \mathsf{e} : \mathsf{T} \mid_{\chi} \mathcal{C}\end{array}}{\Gamma \vdash \mathsf{D}_1 \ldots \mathsf{D}_n\ \mathsf{e} : \mathsf{T} \mid_{\chi \cup \chi_1 \cup \cdots \cup \chi_n} \mathcal{C} \cup \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n} \text{ (CT-Spec)}$$

Fig. 12. Constraint typing relation for $PC_{partial}$.

The constraint typing relation has two important properties on which our unification algorithm relies:

1. In any subtyping constraint of the form $\mathsf{T}_1 <: \mathsf{T}_2$ generated by the constraint typing relation, at least one of T_1, T_2 is *not* a type variable.
2. The rules never add to Γ an assumption of the form $x : X$, where X is a type variable.

We state property 1 formally as Lemma 4.1 below; property 2 is established as part of the proof of Lemma 4.1.

Since subtyping is always defined between base types, Bot and Top, it is clear that $(\mathcal{A}^*, <:)$ forms a complete lattice. Thus the join ($\vee$) and meet ($\wedge$) operators are well defined on $\mathcal{A}^*$. Rule CT-Cond-Base uses the join operator; and both operators are used when we present our unification algorithm in Section 4.3.

We discuss some of the more interesting rules in Fig. 12.

Declaration of variables. Rule CT-DECL ensures that all type variables appearing in type T associated with a declaration are distinct. The expression following the declaration is checked in an extended environment, and the type variables appearing in T form part of the set of used type variables in the conclusion of the rule.

Conditional expressions. In a conditional expression of the form $e_1 \rightarrow e_2 : e_3$ we know that e_1 must have type bool, thus in both rules we post the constraint $\mathsf{T}_1 <: \mathsf{bool}$, where T_1 is the type computed for e_1 by the constraint typing relation. Intuitively, it makes sense to give the result type for the conditional expression the smallest type of which both types T_2 and T_3 are subtypes, where T_2 and T_3 are computed for e_2 and e_3 respectively. This is the join of T_2 and T_3, denoted $\mathsf{T}_2 \vee \mathsf{T}_3$. In general, we cannot compute the join while type checking, since T_2 and T_3 may involve free type variables. To overcome this, we use two rules to handle conditionals.

Rule CT-COND-BASE assumes that T_2 and T_3 are both base types, in which case $\mathsf{T}_2 \vee \mathsf{T}_3$ *can* be computed directly (allowing the case where $\mathsf{T}_2 \vee \mathsf{T}_3 = \mathsf{Top}$, *e.g.* when $\mathsf{T}_2 = \mathsf{bool}$ and $\mathsf{T}_3 = \mathsf{int}$).

Rule CT-COND-COMPOUND assumes that at least one of T_2 and T_3 is *not* a base type. In this case we will show that the non-base types must in fact be channel types, in which case it is sufficient to post the constraint $\mathsf{T}_2 = \mathsf{T}_3$ and give the conditional expression type T_2.

Send, receive and run operations. Rule CT-SEND in Fig. 12 is analogous to the standard typing rule for send operations, rule T-SEND in Fig. 10. The difference is that while T-SEND applies only if the arguments supplied for a send operation are subtypes of the field types for the associated channels, rule CT-SEND always applies, using distinct, fresh type variables to post constraints specifying this subtyping property.

Rules CT-RECEIVE and CT-RUN handle receive and run operations in a similar manner.

4.2.2. Relationship between constraint typing relation and standard type system

Given a $PC_{partial}$ specification S and a PC_{full} type T, if we can find a bounded substitution which, when applied to S, turns S into a PC_{full} specification with type T, then since the substitution does not affect the semantics of S, Theorem 3.2 tells us that S will be free of communication errors.

This is the *declarative* characterization of possible solutions for a specification and a type:

Definition 4.3 (*cf. [26, 22.2.1]*)**.** 1. Let Γ be a context and e an expression. A *solution* for (Γ, e) is a tuple (B, σ, T) such that (B, σ) is a bounded substitution and B; $\sigma(\Gamma) \vdash \sigma(e) : T$.
2. Let Γ be a context and D a declaration. A *solution* for (Γ, D) is a pair (B, σ) such that (B, σ) is a bounded substitution and B; $\sigma(\Gamma) \vdash \sigma(D)$ OK.
3. Let Γ be a context and S a specification. A *solution* for (Γ, S) is a tuple (B, σ, T) such that (B, σ) is a bounded substitution and B; $\sigma(\Gamma) \vdash \sigma(S) : T$.

The declarative characterization does not give a way of finding solutions. However, given a $PC_{partial}$ specification S, the constraint typing relation provides a type U and a set of constraints $\mathcal{C}$, where in general U and $\mathcal{C}$ may share type variables. If a unifier can be computed for $\mathcal{C}$, *i.e.* a bounded substitution (B, σ) satisfying the conditions of Definition 4.2, then the unifier can be applied to U to provide a concrete type T for the specification.

Definition 4.4 (*cf. [26, 22.3.4]*)**.** 1. Let Γ be a context and e an expression. Suppose that $\Gamma \vdash e : \mathsf{U} \mid_\chi \mathcal{C}$. A *solution* for $(\Gamma, e, \mathsf{U}, \mathcal{C})$ is a tuple (B, σ, T) such that (B, σ) is a bounded substitution and (B, σ) satisfies $\mathcal{C}$ and $\sigma(\mathsf{U}) = \mathsf{T}$.
2. Let Γ be a context and D a declaration. Suppose that $\Gamma \vdash D$ OK $\mid_\chi \mathcal{C}$. A *solution* for $(\Gamma, D, \mathcal{C})$ is a pair (B, σ) such that (B, σ) is a bounded substitution and (B, σ) satisfies $\mathcal{C}$.
3. Let Γ be a context and S a specification. Suppose that $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. A *solution* for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ is a tuple (B, σ, T) such that (B, σ) is a bounded substitution and (B, σ) satisfies $\mathcal{C}$ and $\sigma(\mathsf{U}) = \mathsf{T}$.

This *algorithmic* characterization does allow us to find solutions, since we can apply the constraint typing relation algorithmically, and solve constraints using a unification algorithm presented in Section 4.3.

We show that the declarative and algorithmic characterizations are equivalent by showing that every solution for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ is also a solution for (Γ, S), and that every solution for (Γ, S) can be extended to a solution for $(\Gamma, S, \mathsf{U}, \mathcal{C})$.

Theorem 4.1 (*Soundness, cf. [26, 22.3.5]*)**.** 1. *Let Γ be a context and e an expression, and suppose $\Gamma \vdash e : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for $(\Gamma, e, \mathsf{U}, \mathcal{C})$ then it is also a solution for (Γ, e).*
2. *Let Γ be a context and D a declaration, and suppose $\Gamma \vdash D$ OK $\mid_\chi \mathcal{C}$. If (B, σ) is a solution for $(\Gamma, D, \mathcal{C})$ then it is also a solution for (Γ, D).*
3. *Let Γ be a context and S a specification, and suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ then it is also a solution for (Γ, S).*

Definition 4.5 (*cf. [26, 22.3.6]*)**.** $\sigma \setminus \chi$ is the substitution that is undefined for all the variables in χ and otherwise behaves like σ.

Theorem 4.2 (*Completeness, cf. [26, 22.3.7]*)**.** 1. *Let Γ be a context and e an expression, and suppose $\Gamma \vdash e : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for (Γ, e) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', \mathsf{T}')$ for $(\Gamma, e, \mathsf{U}, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$ and $\emptyset \vdash \mathsf{T}' <: \mathsf{T}$.*

2. *Let Γ be a context and D a declaration, and suppose $\Gamma \vdash D \ \mathit{OK} \mid_\chi \mathcal{C}$. If (B, σ) is a solution for (Γ, D) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution (B, σ') for $(\Gamma, D, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$.*
3. *Let Γ be a context and S a specification, and suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for (Γ, S) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', \mathsf{T}')$ for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$ and $\emptyset \vdash \mathsf{T}' <: \mathsf{T}$.*

The previous two theorems establish the following:

Corollary 4.1 (*cf. [26, 22.3.8]*)**.** *Suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. There is a solution for (Γ, S) if and only if there is a solution for $(\Gamma, S, \mathsf{U}, \mathcal{C})$.*

Thus if we can find solutions to the constraints generated by the constraint typing relation for a specification S then we have established that execution of S will not lead to communication errors.

4.3. Unifying constraints

We now turn to the problem of solving a set of subtyping and equality constraints. The basic idea is the Hindley–Milner [15,21] approach of using unification [30] to find the most general solution. In general this is complicated by the presence of subtyping, but our constraint-based typing system for Promela has the property that every subtyping constraint contains at most one type variable. We can exploit this property to define an extension of the standard unification algorithm as presented by Pierce [26, Chapter 22]. The algorithm is given a set of constraints, generated by the constraint typing relation, and either fails or returns the most general bounded substitution which unifies the constraints. In Section 6 we discuss alternative approaches suggested by the literature on type inference with subtyping.

While it would be sufficient, for type safety, merely to show that the set of constraints is satisfiable, obtaining a concrete substitution allows us to present reconstructed types to the user, which aids understanding of Promela specifications. While computing the most general unifier is not strictly necessary (since Promela does not support polymorphic processes), we argue in Section 5.1 that type checking of a sensibly constructed Promela specification should yield a most general unifier where each type variable is assigned to a concrete type; a most general unifier without this property signals a potential flaw in the specification, to which the programmer can be alerted.

4.3.1. The unification algorithm

Our algorithm is presented in Fig. 13. The top-level function *unify* is mutually recursive with the functions *unify_subtype* and *unify_equality*.

The parameters of *unify* are a set $\mathcal{C}$ of constraints and a bounds function B, such that $FV(\mathcal{C}) \subseteq dom(B)$. For a bounds function B and a set $\mathcal{X} \subseteq dom(B)$ of type variables we use $B \setminus \mathcal{X}$ to denote the bounds function B' identical to B except that $B'(X)$ is undefined for each $X \in \mathcal{X}$. For $X \notin dom(B)$ we use $B + [X \mapsto \mathsf{T}]$ to denote the bounds function B' identical to B except that $X \in dom(B')$ and $B'(X) = \mathsf{T}$.

If $\mathcal{C}$ is empty then *unify* returns the bounded substitution $(B, [\ ])$, consisting of the given bounds function and an empty substitution. Otherwise, constraints from $\mathcal{C}$ are processed one at a time, by calling *unify_subtype* or *unify_equality* as appropriate. The order of processing constraints does not matter.

Within *unify_subtype* there are several cases. If the subtyping constraint involves two base types, we simply check that the constraint is satisfied. The next case is for a constraint $X <: \mathsf{T}$, where T is a base type. In this case we adjust the bounds of X by lowering the upper bound so that it is at most T, first checking that the desired new upper bound is not Bot, and solve the remaining constraint under this tighter bounds function. The next case is symmetrical and deals with a constraint $\mathsf{T} <: X$ by increasing the lower bound of X. Finally, a subtyping constraint involving a non-base type is converted into an equality constraint.

Note that the *unify_subtype* function does not consider constraints between type variables (such as $X <: Y$): these are never generated by the constraint typing system:

Lemma 4.1. *Suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If $\mathsf{T}_1 <: \mathsf{T}_2 \in \mathcal{C}$ then at least one of T_1, T_2 is* not *a type variable.*

The function *unify_equality* is similar to the standard unification algorithm, except that bounds must be checked and recomputed when solving constraints involving type variables.

For an equality constraint $\mathsf{T}_1 = \mathsf{T}_2$, if T_1 and T_2 are equal then the constraint is satisfied and can be discarded. The equality check must take into account the possibility that the types are recursive; we discuss how this is handled by our practical implementation in Section 5.2.1.

If T_1 and T_2 are both type variables, X and Y say, we check that the bounds for X and Y are compatible. If so, Y is substituted for X, and the bounds of X are contracted with respect to the bounds for Y.

If T_1 is a type variable X and T_2 is not a type variable then there are two cases, according to whether or not T_2 contains X as a free type variable. If so, then a recursive type is constructed and substituted for X; if not, then T_2 is substituted for X. In both cases we check that T is within the bounds for X. The case where T_2 is a type variable and T_1 is not is handled symmetrically.

Two cases remain: a constraint of the form $\mathsf{chan}\ \mathsf{U}_1 = \mathsf{chan}\ \mathsf{U}_2$ is replaced with the constraint $\mathsf{U}_1 = \mathsf{U}_2$; a constraint of the form $\{\mathsf{U}_1, \ldots, \mathsf{U}_m\} = \{\mathsf{U}'_1, \ldots, \mathsf{U}'_n\}$, when $m = n$, is replaced with m constraints, $\mathsf{U}_i = \mathsf{U}'_i$ $(1 \leq i \leq m)$.

```
function unify(𝒞, B) is
  if 𝒞 = ∅ then
    (B, [ ])
  else
    let T1 ⋈ T2 ∪ 𝒞′ = 𝒞 in
      if ⋈ is <: then
        unify_subtype(T1, T2, 𝒞′, B)
      else
        unify_equality(T1, T2, 𝒞′, B)

function unify_subtype(T1, T2, 𝒞, B) is
  if T1, T2 ∈ 𝒜 then
    if T1 <: T2 then
      unify(𝒞, B)
    else
      fail
  else if T1 = X and T2 ∈ 𝒜 then
    let (LX, UX) = B(X) in
      if LX <: (UX ∧ T2) ≠ Bot then
        unify(𝒞, (B \ {X})+
          [X ↦ (LX, UX ∧ T2)])
      else
        fail
  else if T2 = X and T1 ∈ 𝒜 then
    let (LX, UX) = B(X) in
      if Top ≠ (LX ∨ T1) <: UX then
        unify(𝒞, (B \ {X})+
          [X ↦ (LX ∨ T1, UX)])
      else
        fail
  else
    unify_equality(T1, T2, 𝒞, B)

function unify_equality(T1, T2, 𝒞, B) is
  if T1 = T2 then
    unify(𝒞, B)
  else if T1 = X and T2 = Y then
    let (LX, UX) = B(X) and (LY, UY) = B(Y) in
      if Top ≠ (LX ∨ LY) <: (UX ∧ UY) ≠ Bot then
        unify([Y ↦ X]𝒞, (B \ {X, Y})+
          [X ↦ (LX ∨ LY, UX ∧ UY)])
          ∘[Y ↦ X]
      else
        fail
  else if T1 = X then
    let (LX, UX) = B(X) in
      if LX <: T2 <: UX then
        if T2 = F(X) then
          unify([X ↦ μY.F(Y)]𝒞, B \ {X})
          ∘[X ↦ μY.F(Y)] (where Y is fresh)
        else
          unify([X ↦ T2]𝒞, B \ {X})∘
            [X ↦ T2]
      else
        fail
  else if T2 = X then
    (* Symmetrical *)
  else if T1 = chan U1 and T2 = chan U2 then
    unify(𝒞 ∪ {U1 = U2}, B)
  else if T1 = {U1, ..., Um} and
    T2 = {U′1, ..., U′n}and m = n then
    unify(𝒞 ∪ {U1 = U′1, ..., Um = U′m}, B)
  else
    fail
```

Fig. 13. The unification function.

If the equality constraint does not correspond to one of these forms then unification fails.

If the top-level call $unify(\mathcal{C}, B_0)$ function succeeds in processing all subtyping and equality constraints without failing, then it returns a bounded substitution (B, σ) such that $dom(B) = dom(B_0) \setminus dom(\sigma)$. It is possible for B to be empty, if solving the constraints requires substituting for all of the type variables.

4.3.2. Properties of the unification algorithm

We will show that our unification algorithm can be applied in a way which ensures computation of the most general unifier for a set of constraints, if any unifier exists. To formalise the notion of "most general unifier", we define an order on bounded substitutions.

Definition 4.6. Let (B, σ) be a bounded substitution. An *instance* of (B, σ) is a mapping θ from $dom(B) \cup dom(\sigma)$ to closed type expressions such that

1. for every $X \in dom(B)$ with $B(X) = (L_X, U_X)$, $L_X <: \theta(X) <: U_X$.
2. there exists a substitution γ such that $\theta = \gamma \circ \sigma$ (viewing θ as a substitution).

Definition 4.7 (*cf. [26, 22.4.1]*)**.** A bounded substitution (B_1, σ_1) is *less specific* (or *more general*) than a bounded substitution (B_2, σ_2), written $(B_1, \sigma_1) \sqsubseteq (B_2, \sigma_2)$, if and only if $dom(B_1) \cup dom(\sigma_1) = dom(B_2) \cup dom(\sigma_2)$ and every instance of (B_2, σ_2) is also an instance of (B_1, σ_1).

We now show that the unification procedure of Fig. 13 computes the most general unifier for a set of constraints, if any unifier exists.

Lemma 4.2. *Let $\mathcal{C}$ be a set of constraints and let B_0 be a bounds function such that $FV(\mathcal{C}) \subseteq dom(B_0)$. Then $unify(\mathcal{C}, B_0)$ terminates, either by failing or by returning a result of the form (B, σ).*

Theorem 4.3 (*cf. [26, 22.4.5]*)**.** *Let $\mathcal{C}$ be a set of constraints and let B_0 be the bounds function defined by $dom(B_0) = FV(\mathcal{C})$ and $\forall X \in dom(B_0).B_0(X) = (\mathsf{Bot}, \mathsf{Top})$. If there exists a unifier for $\mathcal{C}$ then $unify(\mathcal{C}, B_0)$ is a unifier for $\mathcal{C}$, and for every unifier (B, σ) for $\mathcal{C}$, $unify(\mathcal{C}, B_0) \sqsubseteq (B, \sigma)$. That is, $unify(\mathcal{C}, B_0)$ is the most general unifier for $\mathcal{C}$.*

We have proved that, for a $PC_{partial}$ specification S, if the constraints generated by the constraint typing relation of Fig. 12 have a solution then *unify* can be used to compute the most general solution. Combining this with Corollary 4.1, this solution can be applied to S to provide a typing under the standard type system of Section 3. This means that, by Theorem 3.2, execution of S does not lead to communication errors.

5. Practical issues

We have proved the correctness of our type checking methods for Promela-calculus, which captures enough of the interesting and relevant features of Promela to provide a rigorous theoretical basis for our practical implementation, ETCH. We now address some practical issues: we show that computing a most general unifier allows us to infer some interesting properties related to channel usage in a specification (Section 5.1), describe the way in which we deal with recursive types (Section 5.2), sketch our approach to handling *conditional* receive statements, which are not part of Promela-calculus (Section 5.3), and address the problem of providing reasonable error messages with type inference (Section 5.4).

5.1. Using most general unifiers for channel usage analysis

Consider the following simple Promela example:

```
chan A = [3] of {int}
chan B = [1] of {byte}
chan C = [1] of {byte}
chan D = [1] of {byte}

proctype P()
{
    A!4;
    B!5;
}

proctype Q()
{
    byte x;
    A?x;
    C?x;
}
```

Messages on channel A have type byte, but we can observe that this message type is wider than necessary: the literal value 4 is regarded as a byte in Promela, and variable x has type byte, thus it would be sufficient for channel A to be declared as: `chan A = [3] of {byte}`. Since int and byte fields require 4 and 1 bytes of storage respectively, and the channel has capacity 3, this modification reduces the space requirements for this channel, *per state*, from 12 to 3 bytes. For a realistic model with tens of millions of states this space reduction would clearly be significant. Note also that in the above specification no message is ever received on channel B, no message is ever sent on C, and no communication whatsoever is performed on D. Such a situation is likely to be unusual, and may signal a mistake in the specification. Nevertheless, the specification is well typed.

Applying ETCH to the above example, invoking an option to look for channel redundancy, automatically discovers these specification flaws. When invoked in this mode, ETCH ignores channel initialisers in the input specification, treating the channels in the above example as if they were declared as follows:

```
chan A
chan B
chan C
chan D
```

ETCH then performs type reconstruction, and reports the following most general types for the channels:

```
A : chan{byte}
B : chan{byte<:X}
C : chan{Y<:byte}
D : chan Z
```

The output for A indicates that A could be safely declared with field type byte rather than int. The output for B shows that B's field type can be any supertype of byte, but has no upper bound: this indicates that no messages are sent on B; similarly the output indicates that no messages are received on C. Finally, D is given type chan *Z*, indicating that D is never used for communication (the arity of D's message tuples is even left unspecified). While this additional information could also be obtained via dataflow analysis [1], it is an interesting by-product of computing a most general unifier.

5.2. Recursive types

Unlike Promela-calculus, Promela does not allow explicit declaration of recursive types, but recursive channel types can be implicitly introduced by channel usage as illustrated by the *client–server* and *telephone* examples of Section 2. Our constraint-based type checking algorithm may generate recursive types during unification (see Section 4.3).

Although the user never writes recursive type expressions, they may encounter them in error messages, or in the reconstructed type information which ETCH generates. We describe the techniques ETCH uses to compare recursive type expressions, and to minimise recursive types so that they are presented to the user in as palatable a form as possible.

```
chan A = [1] of {chan};
chan B = [1] of {chan};
chan C = [1] of {chan};
chan D = [1] of {chan,chan,chan};
chan E = [1] of {chan,chan,chan};
chan F = [1] of {byte}

init {
  A!B;
  B!C;
  C!A;
  D!E,E,A;
  E!D,E,B;
  F!E
}
```

Fig. 14. A contrived Promela specification which generates a large recursive type expression if minimisation is not applied.

5.2.1. Comparing recursive types

The *unify_equality* procedure (Fig. 13) in our unification algorithm (Section 4.3) takes types T_1 and T_2, and checks whether $\mathsf{T}_1 = \mathsf{T}_2$. This check for structural equivalence of types needs to handle the case where T_1 and/or T_2 are recursive.

To handle recursive types in a straightforward manner our uses techniques from a unification algorithm given in [1]. Types are represented as directed graphs with a cyclic graph corresponding to a recursive type. Each graph node n has a representative $rep(n)$, a pointer to another graph node. Initially each node n represents itself, *i.e.* $rep(n) = n$. During unification, representatives are re-assigned. If unification succeeds then the substitution part of the bounded substitution (B, σ) is the mapping defined by, for each type variable X, $X \mapsto rep^*(X)$, where, for a node n, $rep^*(n)$ is computed by following rep pointers and returning the node m eventually reached for which $rep(m) = m$.

Representatives are computed as follows. When two chan nodes are unified via a constraint $\mathsf{chan}\ \mathsf{U}_1 = \mathsf{chan}\ \mathsf{U}_2$, the representative for the node associated with the left hand side of the constraint is set to the representative for the right hand node. Product type nodes are handled similarly when solving constraints of the form $\{\mathsf{U}_1, \ldots \mathsf{U}_m\} = \{\mathsf{U}'_1, \ldots, \mathsf{U}'_m\}$. When two type variables are unified via a constraint $X = Y$, $rep(X)$ is set to $rep(Y)$. When a type variable is unified with a non-type variable, the representative for the type variable node is set to the representative for the non-type variable node. The substitution of a type variable for a recursive type, specified as $X \mapsto \mu Y.F(Y)$ in Fig. 13, is handled implicitly by introducing a cycle in the type graph: we do not use an explicit μ constructor.

To check equality of types T_1 and T_2, it suffices for our algorithm to check whether $rep^*(n_1) = rep^*(n_2)$, where n_1 and n_2 are the type graph nodes associated with T_1 and T_2 respectively. The types may not be identified as equal straight away: when we first compare T_1 with T_2 we may just set $rep(n_1)$ to $rep(n_2)$. The next time the types are compared (when the unification processes has explored the cyclic type structures sufficiently) we are guaranteed to have $rep^*(n_1) = rep^*(n_2)$.

5.2.2. Recursive type minimisation

A recursive type can have infinitely many forms. Consider the type expression $\mathsf{rec}X.\mathsf{chan}\{X, \mathsf{mtype}\}$ associated with the *client–server* specification of Section 2.1. Alternative ways to write this type include:

- $\mathsf{chan}\{\mathsf{rec}X.\mathsf{chan}\{X, \mathsf{mtype}\}, \mathsf{mtype}\}$
- $\mathsf{chan}\ \mathsf{rec}X.\{\mathsf{chan}\ X, \mathsf{mtype}\}$
- $\mathsf{chan}\{\mathsf{chan}\{\mathsf{chan}\{\mathsf{chan}\{\mathsf{chan}\{\mathsf{rec}X.\mathsf{chan}\{X, \mathsf{mtype}\}, \mathsf{mtype}\}, \mathsf{mtype}\}, \mathsf{mtype}\}, \mathsf{mtype}\}, \mathsf{mtype}\}$

One issue which motivates techniques for recursive type minimisation is that of storage: in a production compiler it is desirable to store type expressions as compactly as possible to avoid excessive memory overhead during compilation. Sophisticated minimisation techniques for recursive types have been developed for this purpose [5,20]. Our motivation is to improve the readability of messages generated by Etch. The aim of an enhanced type checker is to assist with compile-time debugging, for which good quality error messages are essential.

To illustrate the improvement in output quality which type minimisation affords, consider the variables `pool` and `partner` from the *client–server* and *telephone* specifications respectively (see Sections 2.1 and 2.2). Fig. 4 shows the complete types which Etch reconstructs for these variables.

If we run Etch without type minimisation then the reconstructed type information for these variables is presented as follows:

```
pool    : chan { chan rec X . { mtype, chan X } }
partner : array(size 3) of chan rec X . { chan X, bit }
```

In each case, the displayed type expression is larger than need be. For a more extreme case where type minimisation can drastically improve readability, consider the contrived Promela example of Fig. 14.

$$\frac{x : \mathsf{T} \in \Gamma \qquad x_i : \mathsf{T}_i \in \Gamma \qquad \Gamma \vdash \mathsf{e}_j : \mathsf{T}_j \mid_\chi \mathcal{C} \qquad X_j \neq X_i \notin \chi, \mathsf{T}_i, \mathsf{T}_j, \mathcal{C}, \Gamma, \mathsf{e}_j \qquad \mathcal{C}' = \{\mathsf{T} = \mathsf{chan}\{\ldots, X_i, \ldots, X_j, \ldots\}, X_i <: \mathsf{T}_i, \ldots, \mathsf{T}_j <: X_j\} \cup \mathcal{C}}{\Gamma \vdash x\,?\,\ldots, x_i, \ldots, \mathtt{eval}(\mathsf{e}_j), \ldots : \mathsf{Unit} \mid_{\{X_i, X_j\} \cup \chi} \mathcal{C}'} \text{ (CT-Cond-Receive)}$$

Fig. 15. Typing rule illustrating the way conditional receive operations are handled by Etch. The rule shows an example of a variable argument and an **eval** argument. For ease of presentation, details of constraints and type variables for the other arguments are omitted.

The code fragment is not well typed: the channel F accepts single field byte messages, but the channel E is supplied as an argument to F at line 14. Etch reports this via a reasonable error message:

```
Error at line 14: "rec X.chan{X,X,rec Y.chan {Y}}" is not compatible
with type "byte".
```

When minimisation is turned off the error message becomes much more difficult to understand:

```
Error at line 14: "rec Z.chan rec X.{chan {chan X,chan X,chan rec
Y.{chan{chan{chan Y}}}}, Z,chan rec A.{chan{chan{chan A}}}}" is not
compatible with type "byte".
```

The Etch recursive type minimisation algorithm is based on an algorithm for minimisation of deterministic finite automata (DFA) given in [19]. There are many algorithms for DFA minimisation in the literature (see [35] for a taxonomy). We settled on the algorithm of [19] for its simplicity and ease of correct implementation.

5.3. Type checking full Promela

Promela-calculus is deliberately much simpler than Promela, but from the perspective of implementing a type checker, most of the additional features of Promela can be handled easily. We briefly discuss one feature of full Promela which required some consideration when designing Etch: constraint typing for *conditional* receive statements.

A useful feature of Promela is the ability to *conditionally* receive messages on a channel. In the following example:

```
chan c = [0] of {int};
...
byte x = ...;
c?eval(x)
```

the statement `c?eval(x)` blocks unless an integer message is offered on channel `c` such that the value of the message equals the runtime value of `x`. The conditional receive does not modify variables of the recipient; the only side effect is that a message is removed from the channel, if buffered. Note that an arbitrary expression, not just a variable, can appear as an argument to **eval**.

Without the **eval** construct, we would deem the above example ill-typed, as it attempts to receive an int argument into a byte variable. However, with the **eval** construct it does not make sense to reject this example: the receive statement requires a value of type byte, and since the channel communicates int values, which contain all byte values, the statement will be executable if the correct int value is transmitted.

On the other hand, we could regard the following example as ill-typed:

```
chan c = [0] of {byte};
...
int x = ...;
c?eval(x)
```

The receive statement requires that some value of type int be available on a channel which transmits values of the smaller type byte. Suppose that `x` has value -1 at runtime, which is an int but not a byte. Then the receive statement will *never* succeed, which could be viewed as a programming error.

Based on this discussion, we reverse the way in which subtyping constraints are posted for **eval** receive arguments. Recall that when the ith argument to a receive operation is a variable with type T_i, we choose a fresh type variable X_i for the argument type, and post the constraint $X_i <: \mathsf{T}_i$ (see rule CT-Receive in Fig. 12). When the ith argument has the form **eval**(e), where e has type T_i, we again choose a fresh type variable X_i, but instead post the constraint $T_i <: X_i$. This captures the idea that a conditional receive construct should expect values belonging to a subset of the values which it is possible to send on a channel. Type rule CT-Cond-Receive in Fig. 15 describes the way constraints are posted when both variable and **eval** arguments are used in a receive statement.

5.4. Providing reasonable error messages

Provision of informative error messages is an important feature of any practical type checker. Error reporting for standard type checking is relatively easy: a type error typically corresponds to a direct misuse of types, which can be isolated to a single line of source code. In theoretical terms, this corresponds to an expression for which no type rule applies. Error reporting with type inference is harder. The constraint typing relation does not fail: every expression in a program is successfully assigned a type, and type errors are reported if *unification* fails. It is difficult to relate a unification error to a specific line of source code, since the error may result from unification of several constraints, arising from distinct areas of the program. In the design of Etch, we use two simple techniques to provide reasonable error messages.

First, we do not implement the constraint typing relation strictly: when checking a compound expression, if concrete types are available for all sub-expressions then we apply standard type checking, logging a type error immediately if type checking fails. For example, if x : bit and y : int, on encountering the assignment $x = y$, rather than posting constraint bit $<:$ int we immediately report a subtyping error, with a message tailored to the fact that the subtyping error occurs in the context of an assignment. Simply posting the constraint bit $<:$ int would lead to a subtyping error during unification, but without passing significant additional information to the unifier, we would not know the context in which the subtyping error had occurred, so could not give such a good error message.

Secondly, we associate with each constraint the source code line at which the constraint was generated. When a constraint causes unification to fail, this line number is used for error reporting. This is better than nothing: often a constraint corresponds to a direct type error at the point of constraint generation. However, if several incompatible constraints are generated, *e.g.* according to the way in which a channel is used, the specific constraint which causes unification to fail may result from *correct* channel usage, if constraints arising from incorrect channel usage have been processed first. In this case, a type error will be generated referring to a seemingly correct line of source code, leaving the user to investigate the real cause of the error.

6. Related work

The Etch type checker, together with an informal description of our approach to type checking, are described in a tool demonstration paper [8]. To our knowledge, there is no other work which concentrates directly on type checking for Promela. A paper introducing *Promela+*, an extension of Promela for timed interactive simulation [6], describes a software tool which claims to provide better type checking than standard Spin, though no further details on the nature of this type checking are provided.

The problem of type inference in the presence of subtyping has been studied by several authors, including [33,27,2,9,28, 23,34,14]. We can observe that our approach, of finding the most general unifier as a bounded substitution, gives more than we really need for the specific problem of type checking Promela. It would be sufficient to solve the equality constraints in the standard way, apply the resulting substitution to the subtyping constraints, and then check satisfiability of the resulting subtyping constraints. Tiuryn [33] has shown that satisfiability can be decided in polynomial time, although his decision procedure does not construct an explicit solution of the constraints and does not reveal the bounds of each type variable in the way that our algorithm does. In practice, an explicit substitution allows us to present the modeller with reconstructed types for channels in a Promela specification, which aids program understanding. Furthermore, computing a most general substitution may reveal additional properties of interest related to channel usage, as discussed in Section 5.1.

The theory HM(X) [23] allows the Hindley–Milner system to be parameterised by a constraint system. The theory gives conditions for principal typings to exist, but does not automatically provide an algorithm for solving constraints. We believe that our system can be expressed as an instance of HM(X), although we have not checked this in detail, but in any case we would still need to define an algorithm for solving subtyping constraints. Pottier [27] has also studied type inference with subtyping, focusing on efficient algorithms, although the algorithms themselves are not defined very explicitly. We have not made a detailed comparison with his work, as we found it easier to develop our own algorithm based on the specific properties of our constraint sets. However, in relation to efficiency, we can make two observations. First, our implementation uses the technique, crucial for making unification efficient, of representing type expressions by directed graphs. Second, Promela does not include let-polymorphism, which is the cause of potential inefficiency in ML-style type inference. In practice we have found our implementation to be acceptably efficient.

We note that the most general solutions produced by our algorithm could be used as the basis for a system of ML-style polymorphic typing for Promela, either to allow polymorphic process definitions or to support separate type checking of Promela code in multiple files. Our algorithm might be of interest for other languages with similar properties to Promela.

Sophisticated techniques have been developed for generating good error messages with type inference for functional languages: see [13] and references therein, as well as a recent tool for type error slicing in standard ML [29,11]. Although we have found the simple methods for error reporting discussed in Section 5.4 to work reasonably well in practice, the usability of Etch could potentially be improved by implementing a more advanced system of error tracking.

7. Conclusions and future work

We have presented details of the design and implementation of Etch, an enhanced type checker for Promela. Etch is able to statically detect errors which are beyond the limited scope of type checking performed by the Spin model checker, thus Etch can aid the development of robust Promela specifications for formal verification. Etch is publicly available for use by the Promela/Spin community.

As discussed in Section 2.3, an avenue for future work is to extend Etch with session types, to widen the class of specifications that can be successfully type checked. The main challenge here would be inferring session types from the way in which channels are used, so that Etch can continue to be applied to Promela specifications without requiring additional type annotations.

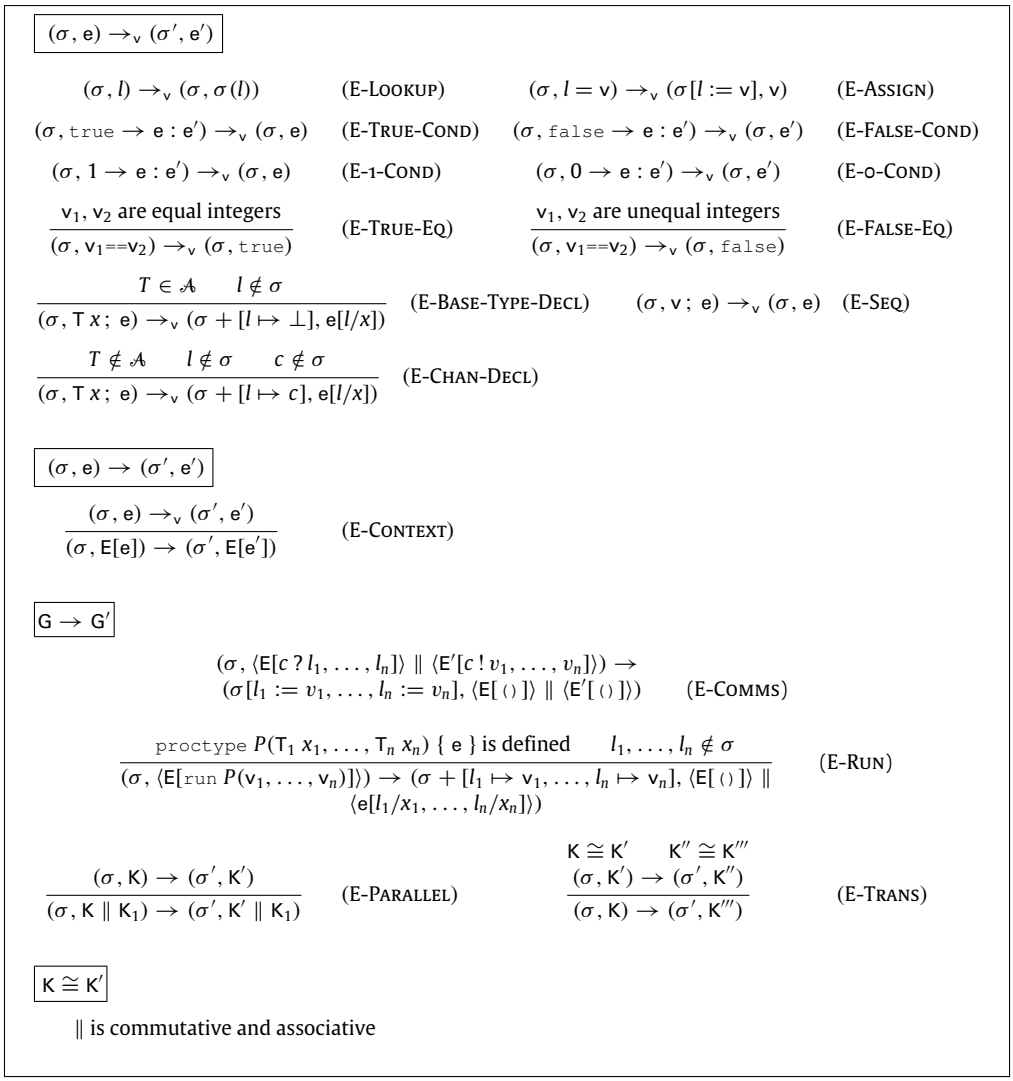

Fig. 16. Operational semantics for Promela-calculus.

Acknowledgements

Alastair F. Donaldson was supported by a Ph.D. studentship from the Carnegie Trust for the Universities of Scotland, and subsequently by the grant "Advanced Formal Verification Techniques for Heterogeneous Multi-core Programming" (EP/G051100/1) from the UK Engineering and Physical Sciences Research Council (EPSRC). Simon Gay was partially supported by the grant "Engineering Foundations of Web Service: Theories and Tool Support" (EP/E065708/1) from the UK Engineering and Physical Sciences Research Council (EPSRC).

Appendix A. Operational semantics for Promela-calculus

Operational semantics for *PC* are presented in Fig. 16.

Rules of the form $(\sigma, \mathsf{e}) \to_{\mathsf{v}} (\sigma', \mathsf{e}')$ show how expressions are evaluated with respect to a store. The E-Lookup and E-Assign rules show respectively how the store σ is referenced to look up the value of a store location, and updated when a store location is written to. Evaluation of conditional expressions are handled naturally by rules E-True-Cond and E-False-Cond, while analogous rules E-1-Cond and E-0-Cond show how the bit literals 1 and 0 can be used in place of `true` and `false`. Rules E-True-Eq and E-False-Eq allow numeric values to be compared to yield a boolean result, and rule E-Seq shows how sequences of statements are evaluated. Declaration of variables with base types is handled by the E-Base-Type-Decl rule, which picks a fresh store location for each such declaration. When a variable is declared with a type $\mathsf{T} \notin \mathcal{A}$, rule E-Chan-Decl

introduces a new channel value. We introduce type rules in Section 3.3 such that for any type T used in a variable declaration in a well-typed specification, either $\mathsf{T} \in \mathcal{A}$ or T is a (possibly recursive) channel type.

The evaluation context syntax of Fig. 9, together with the E-Context rule, allow more complex forms of expressions to be evaluated in terms of rules of the form $(\sigma, \mathsf{e}) \rightarrow_{\mathsf{v}} (\sigma', \mathsf{e}')$. For example, the pair of evaluation context $\mathsf{E}\,!\,\mathsf{e}, \ldots, \mathsf{e}$ and $\mathsf{v}\,!\,\mathsf{v}, \ldots, \mathsf{v}, \mathsf{E}, \mathsf{e}, \ldots, \mathsf{e}$ specify that the expression to the left of a ! operator should be evaluated to a value before the expressions on the right are evaluated, and that these expressions should be evaluated in left-to-right order. The other forms of evaluation context are similar.

Semantics for communication are provided by rule E-Comms. The rule states that if a thread is ready to send n values on channel c and another thread is ready to receive into n store locations, also via c, then the store should be updated so that each location l_i is set to the corresponding value v_i ($1 \leq i \leq n$). The send and receive operations reduce to the `()` literal value of type Unit. Rule E-Run provides call-by-value semantics for proctype instantiation.

The remaining rules provide algebraic identities which allow parallel compositions of threads to be maneuvered in order for an evaluation rule to apply.

Appendix B. Proofs omitted from the text

For convenience, we re-state each result.

B.1 Proofs of results in Section 4.2

Theorem 4.1 (soundness)

1. Let Γ be a context and e an expression, and suppose $\Gamma \vdash e : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for $(\Gamma, e, \mathsf{U}, \mathcal{C})$ then it is also a solution for (Γ, e).
2. Let Γ be a context and D a declaration, and suppose $\Gamma \vdash D \ \mathtt{OK} \mid_\chi \mathcal{C}$. If (B, σ) is a solution for $(\Gamma, D, \mathcal{C})$ then it is also a solution for (Γ, D).
3. Let Γ be a context and S a specification, and suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ then it is also a solution for (Γ, S).

Proof. In each case, a straightforward induction on the typing derivation, with a case analysis on the last rule. We give details for the rules CT-Cond-Base and CT-Send in case 1.

(CT-Cond-Base): In this case e is $\mathsf{e}_1 \rightarrow \mathsf{e}_2 : \mathsf{e}_3$ and from the hypotheses of the instance of CT-Cond-Base we have $\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid \mathcal{C}_1, \Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid \mathcal{C}_2, \Gamma \vdash \mathsf{e}_3 : \mathsf{T}_3 \mid \mathcal{C}_3$, and $\mathsf{T}_2, \mathsf{T}_3 \in \mathcal{A}^*$, with $\mathsf{U} = \mathsf{T}_2 \vee \mathsf{T}_3$ and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}\}$. We omit the χ annotations as they are not used in this proof.

Because (B, σ, T) is a solution for $(\Gamma, \mathsf{e}_1 \rightarrow \mathsf{e}_2 : \mathsf{e}_3, \mathsf{U}, \mathcal{C})$, we have that (B, σ) satisfies $\mathcal{C}$ and $\sigma(\mathsf{U}) = \mathsf{T}$. Therefore (B, σ) satisfies $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$, as well as $\mathsf{T}_1 <: \mathsf{bool}$, meaning that $B; \sigma(\mathsf{T}_1) <: \mathsf{bool}$.

From the algorithmic definition of a solution (Definition 4.4), $(B, \sigma, \sigma(\mathsf{T}_1))$ is a solution for $(\Gamma, \mathsf{e}_1, \mathsf{T}_1, \mathcal{C}_1)$, and similarly for e_2 and e_3. Applying the induction hypothesis, $(B, \sigma, \sigma(\mathsf{T}_1))$ is a solution for (Γ, e_1) in the declarative sense (Definition 4.3), *i.e.* $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) : \sigma(\mathsf{T}_1)$ is derivable in the standard type system. The same reasoning applies to e_2 and e_3.

Because T_2 and T_3 are base types we have $\sigma(\mathsf{T}_2) = \mathsf{T}_2$ and $\sigma(\mathsf{T}_3) = \mathsf{T}_3$. It follows that, $\mathsf{U} = \mathsf{T}_2 \vee \mathsf{T}_3$ is either a base type or Top and so $\sigma(\mathsf{U}) = \mathsf{U}$; hence $\mathsf{U} = \mathsf{T} = \mathsf{T}_2 \vee \mathsf{T}_3$.

From $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) : \sigma(\mathsf{T}_1)$ and $B; \sigma(\mathsf{T}_1) <: \mathsf{bool}$ we can derive $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) : \mathsf{bool}$ by using rule T-Sub. Similarly, since $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_2) : \sigma(\mathsf{T}_2), \sigma(\mathsf{T}_2) = \mathsf{T}_2$ and $\mathsf{T}_2 <: \mathsf{T}_2 \vee \mathsf{T}_3$, we have $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_2) : \mathsf{T}_2 \vee \mathsf{T}_3$. By a symmetric argument, $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_3) : \mathsf{T}_2 \vee \mathsf{T}_3$. Finally, rule T-Cond gives $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) \rightarrow \sigma(\mathsf{e}_2) : \sigma(\mathsf{e}_3) : \mathsf{T}_2 \vee \mathsf{T}_3$. Because $\sigma(\mathsf{e}_1) \rightarrow \sigma(\mathsf{e}_2) : \sigma(\mathsf{e}_3) = \sigma(\mathsf{e}_1 \rightarrow \mathsf{e}_2 : \mathsf{e}_3)$ and $\mathsf{T}_2 \vee \mathsf{T}_3 = \mathsf{T}$, this means that (B, σ, T) is a solution for (Γ, e), as required.

(CT-Send): In this case e is $x!\mathsf{e}_1, \ldots, \mathsf{e}_n$. From the hypothesis of the instance of CT-Send we have $x : \mathsf{T}' \in \Gamma, \Gamma \vdash \mathsf{e}_i : \mathsf{T}_i \mid \mathcal{C}_i$ ($1 \leq i \leq n$), $\mathsf{U} = \mathsf{Unit}$ and $\mathcal{C} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{T}' = \mathsf{chan}\{X_1, \ldots, X_n\}, \mathsf{T}_1 <: X_1, \ldots, \mathsf{T}_n <: X_n\}$. Again, we omit the χ annotations.

Because (B, σ, T) is a solution for $(\Gamma, x!\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{U}, \mathcal{C})$ we have (B, σ) satisfies $\mathcal{C}$ and $\sigma(\mathsf{U}) = \mathsf{T}$. Therefore (B, σ) satisfies $\mathcal{C}_1, \ldots, \mathcal{C}_n$ as well as $\mathsf{T}' = \mathsf{chan}\{X_1, \ldots, X_n\}$ and $\mathsf{T}_i <: X_i$ ($1 \leq i \leq n$). Since $\mathsf{U} = \mathsf{Unit}$, $\mathsf{T} = \sigma(\mathsf{U}) = \mathsf{Unit}$.

For $1 \leq i \leq n$, combining the fact that $(B, \sigma, \sigma(\mathsf{T}_i))$ is a solution for $(\Gamma, \mathsf{e}_i, \mathsf{T}_i, \mathcal{C}_i)$ with the induction hypothesis, we have $(B, \sigma, \sigma(\mathsf{T}_i))$ is a solution for (Γ, e_i), *i.e.* $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_i) : \sigma(\mathsf{T}_i)$ is derivable in the original type system. Since (B, σ) satisfies $\mathsf{T}_i <: X_i$ we have $B \vdash \sigma(\mathsf{T}_i) <: \sigma(X_i)$. Using rule T-Sub gives $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_i) : \sigma(X_i)$.

We have $x : \mathsf{T}' \in \Gamma$, so $x : \sigma(\mathsf{T}') \in \sigma(\Gamma)$. Now (B, σ) satisfies $T' = \mathsf{chan}\{X_1, \ldots, X_n\}$, so $\sigma(\mathsf{T}') = \sigma(\mathsf{chan}\{X_1, \ldots, X_n\}) = \mathsf{chan}\{\sigma(X_1), \ldots, \sigma(X_n)\}$. It follows that $x : \mathsf{chan}\{\sigma(X_1), \ldots, \sigma(X_n)\} \in \sigma(\Gamma)$.

Rule T-Send now applies, to yield $B; \sigma(\Gamma) \vdash x!\sigma(\mathsf{e}_1), \ldots, \sigma(\mathsf{e}_n) : \mathsf{Unit}$. Because $x!\sigma(\mathsf{e}_1), \ldots, \sigma(\mathsf{e}_n) = \sigma(x!\mathsf{e}_1, \ldots, \mathsf{e}_n)$ and $\mathsf{Unit} = \mathsf{T}$, this means (B, σ, T) is a solution for (Γ, e) as required. □

Theorem 4.2 (completeness)

1. Let Γ be a context and e an expression, and suppose $\Gamma \vdash e : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for (Γ, e) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', \mathsf{T}')$ for $(\Gamma, e, \mathsf{U}, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$ and $\emptyset \vdash \mathsf{T}' <: \mathsf{T}$.

2. Let Γ be a context and D a declaration, and suppose $\Gamma \vdash D \text{ OK} \mid_\chi \mathcal{C}$. If (B, σ) is a solution for (Γ, D) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution (B, σ') for $(\Gamma, D, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$.
3. Let Γ be a context and S a specification, and suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If (B, σ, T) is a solution for (Γ, S) and $dom(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', \mathsf{T}')$ for $(\Gamma, S, \mathsf{U}, \mathcal{C})$ such that $\sigma' \setminus \chi = \sigma$ and $\emptyset \vdash \mathsf{T}' <: \mathsf{T}$.

Proof. Again, the proof is by straightforward induction on typing derivations. We give details for the rules CT-Cond-Compound, CT-Send and CT-Receive in case 1.

(CT-Cond-Compound): In this case $e = \mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3$, $\mathsf{U} = \mathsf{T}_2$, $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}, \mathsf{T}_2 = \mathsf{T}_3\}$.

From the premises of the rule we have $\Gamma \vdash \mathsf{e}_1 : \mathsf{T}_1 \mid_{\chi_1} \mathcal{C}_1$, $\Gamma \vdash \mathsf{e}_2 : \mathsf{T}_2 \mid_{\chi_2} \mathcal{C}_2$, $\Gamma \vdash \mathsf{e}_3 : \mathsf{T}_3 \mid_{\chi_3} \mathcal{C}_3$. We also have $\mathsf{T}_2 \notin \mathcal{A}^*$ *or* $\mathsf{T}_3 \notin \mathcal{A}^*$.

Since (B, σ, T) is a solution for $(\Gamma, \mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3)$ we have $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3) : \mathsf{T}$, *i.e.* $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) \to \sigma(\mathsf{e}_2) : \sigma(\mathsf{e}_3) : \mathsf{T}$. Inverting rule T-Cond tells us:

- $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_1) : \mathsf{bool}$
- $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_2) : \mathsf{T}$
- $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_3) : \mathsf{T}$

This means that $(B, \sigma, \mathsf{bool})$ is a solution for (Γ, e_1), and (B, σ, T) is a solution for both (Γ, e_2) and (Γ, e_3). By the induction hypothesis there are solutions $(B, \sigma_1, \mathsf{K}_1)$ for $(\Gamma, \mathsf{e}_1, \mathsf{T}_1, \mathcal{C}_1)$, $(B, \sigma_2, \mathsf{K}_2)$ for $(\Gamma, \mathsf{e}_2, \mathsf{T}_2, \mathcal{C}_2)$ and $(B, \sigma_3, \mathsf{K}_3)$ for $(\Gamma, \mathsf{e}_3, \mathsf{T}_3, \mathcal{C}_3)$ such that $\sigma_1 \setminus \chi_1 = \sigma, \sigma_2 \setminus \chi_2 = \sigma, \sigma_3 \setminus \chi_3 = \sigma, \emptyset \vdash \mathsf{K}_1 <: \mathsf{bool}, \emptyset \vdash \mathsf{K}_2 <: \mathsf{T}$ and $\emptyset \vdash \mathsf{K}_3 <: \mathsf{T}$.

Because for $1 \le i \ne j \le 3$ we have $\sigma_i \setminus \chi_i = \sigma \setminus \chi_j$ and $\chi_i \cap \chi_j = \emptyset$ we can extend $\sigma_1, \sigma_2, \sigma_3$ to σ' such that $\sigma' \setminus \chi_1 \cup \chi_2 \cup \chi_3 = \sigma$. To complete the proof for case CT-Cond-Compound we show that (B, σ', T) is a solution for $(\Gamma, \mathsf{e}_1 \to \mathsf{e}_2 : \mathsf{e}_3, \mathsf{T}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}, \mathsf{T}_2 = \mathsf{T}_3\})$.

We need to show that σ' satisfies $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\mathsf{T}_1 <: \mathsf{bool}, \mathsf{T}_2 = \mathsf{T}_3\}$. We have σ' satisfies $\mathcal{C}_i$ from the fact that σ_i satisfies $\mathcal{C}_i$ $(1 \le i \le 3)$. Since T_1 does not include any type variables contained in χ_1, $\sigma'(\mathsf{T}_1) = \sigma_1(\mathsf{T}_1) = \mathsf{K}_1 <: \mathsf{bool} = \sigma'(\mathsf{bool})$, so σ' satisfies $\mathsf{T}_1 <: \mathsf{bool}$. For $i \in \{2, 3\}$, no variable in χ_i appears in T_i, so $\sigma'(\mathsf{T}_i) = \sigma_i(\mathsf{T}_i)$. So we have $\sigma'(\mathsf{T}_2) = \mathsf{K}_2$ and $\sigma'(\mathsf{T}_3) = \mathsf{K}_3$. We know that at least one of $\mathsf{T}_2, \mathsf{T}_3$ is not a base type. It follows that at least one of $\sigma'(\mathsf{T}_2), \sigma'(\mathsf{T}_3)$, *i.e.* one of $\mathsf{K}_2, \mathsf{K}_3$, is not a base type. Since subtyping can only occur between base types, $\emptyset \vdash \mathsf{K}_2 <: \mathsf{T}$ and $\emptyset \vdash \mathsf{K}_3 <: \mathsf{T}$ implies that $\mathsf{K}_2 = \mathsf{K}_3 = \mathsf{T}$. Thus σ' satisfies $\mathsf{T}_2 = \mathsf{T}_3$.

(CT-Send): In this case $e = x!\mathsf{e}_1, \ldots, \mathsf{e}_n$, $\mathsf{U} = \mathsf{Unit}$, $\mathcal{C} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, \mathsf{T}_1 <: X_1, \ldots, \mathsf{T}_1 <: X_n\}$. From the premises of the rule we have: $x : \mathsf{V} \in \Gamma$, $\Gamma \vdash \mathsf{e}_i : \mathsf{T}_i \mid_{\chi_i} \mathcal{C}_i$ $(1 \le i \le n)$.

Since (B, σ, T) is a solution for $(\Gamma, (x!\mathsf{e}_1, \ldots, \mathsf{e}_n))$ we have $B; \sigma(\Gamma) \vdash \sigma(x!\mathsf{e}_1, \ldots, \mathsf{e}_n) : \mathsf{T}$, *i.e.* $B; \sigma(\Gamma) \vdash x!\sigma(\mathsf{e}_1), \ldots, \sigma(\mathsf{e}_n) : \mathsf{T}$. Inverting rule T-Send we have:

- $\mathsf{T} = \mathsf{Unit}$
- $B; \sigma(\Gamma) \vdash \sigma(\mathsf{e}_i) : \mathsf{W}_i$ $(1 \le i \le n)$
- $x : \mathsf{chan}\{\mathsf{W}_1, \ldots, \mathsf{W}_n\} \in \sigma(\Gamma)$, *i.e.* $x : \mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n\} \in \Gamma$, where $\sigma(\mathsf{V}_i) = \mathsf{W}_i$ $(1 \le i \le n)$.

This tells us that $\mathsf{V} = \mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ and $(B, \sigma, \mathsf{W}_i)$ is a solution for (Γ, e_i) $(1 \le i \le n)$.

For each i $(1 \le i \le n)$ by the induction hypothesis there is a solution $(B, \sigma_i, \mathsf{K}_i)$ for $(\Gamma, \mathsf{e}_i, \mathsf{T}_i, \mathcal{C}_i)$ such that $\sigma_i \setminus \chi_i = \sigma$ and $\emptyset \vdash \mathsf{K}_i <: \mathsf{W}_i$. Therefore (B, σ_i) satisfies $\mathcal{C}_i$ and $\sigma_i(\mathsf{T}_i) = \mathsf{K}_i$.

Because for any $1 \le i \ne j \le n$, $\sigma_i \setminus \chi_i = \sigma_j \setminus \chi_j$ and $\chi_i \cap \chi_j = \emptyset$, we can extend $\sigma_1, \ldots, \sigma_n$ to σ' such that $\sigma' \setminus \chi_1 \cup \cdots \cup \chi_n \cup \{X_1, \ldots, X_n\} = \sigma$ and $\sigma'(X_i) = W_i$ $(1 \le i \le n)$. To complete the proof for CT-Send we must show that $(B, \sigma', \mathsf{Unit})$ is a solution for $(\Gamma, (x!\mathsf{e}_1, \ldots, \mathsf{e}_n), \mathsf{Unit}, \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, \mathsf{T}_1 <: X_1, \ldots, \mathsf{T}_n <: X_n\})$. This requires σ' to satisfy $\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n \cup \{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, \mathsf{T}_1 <: X_1, \ldots, \mathsf{T}_n <: X_n\}$. For $1 \le i \le n$, σ' satisfies $\mathcal{C}_i$ due to the fact that σ_i satisfies $\mathcal{C}_i$. Since neither X nor any type variable in χ_i is contained in T_i, $\sigma'(\mathsf{T}_i) = \sigma_i(\mathsf{T}_i) = \mathsf{K}_i <: \mathsf{W}_i = \sigma'(X_i)$, so σ' satisfies $\mathsf{T}_i <: X_i$. Finally, $\sigma'(\mathsf{V}) = \sigma'(\mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n\}) = \mathsf{chan}\{(\sigma'(\mathsf{V}_1), \ldots, \sigma'(\mathsf{V}_n)\} = \mathsf{chan}\{(\sigma(\mathsf{V}_1), \ldots, \sigma(\mathsf{V}_n)\}$ (since neither X nor any type variable in $\chi_1 \cup \cdots \cup \chi_n$ is contained in any V_i $(1 \le i \le n)$) $= \mathsf{chan}\{\sigma'(X_1), \ldots, \sigma'(X_n)\} = \sigma(\mathsf{chan}\{X_1, \ldots, X_n\})$.

(CT-Receive): In this case $e = x?x_1, \ldots, x_n$, $U = \mathsf{Unit}$, $\mathcal{C} = \{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, X_1 <: \mathsf{T}_1, \ldots, X_n <: \mathsf{T}_n\}$. From the premises of the rule we have $x : \mathsf{V} \in \Gamma$ and $x_i : \mathsf{T}_i \in \Gamma$ $(1 \le i \le n)$.

Since (B, σ, T) is a solution for $(\Gamma, (x?x_1, \ldots, x_n))$ we have $B; \sigma(\Gamma) \vdash \sigma(x?x_1, \ldots, x_n) : \mathsf{T}$, *i.e.* $B; \sigma(\Gamma) \vdash x?x_1, \ldots, x_n : \mathsf{T}$. Inverting the T-Receive rule tells us:

- $\mathsf{T} = \mathsf{Unit}$
- $x : \mathsf{chan}\{\mathsf{W}_1, \ldots, \mathsf{W}_n\} \in \sigma(\Gamma)$, *i.e.* $x : \mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n\} \in \Gamma$, where $\sigma(\mathsf{V}_i) = \mathsf{W}_i$ $(\le i \le n)$
- $x_i : \mathsf{U}_i \in \sigma(\Gamma)$ $(1 \le i \le n)$
- $B; \mathsf{W}_i <: \mathsf{U}_i$ $(1 \le i \le n)$

Therefore we have $\mathsf{V} = \mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n, \sigma(\mathsf{T}_i) = \mathsf{U}_i$ and $B; \sigma(\mathsf{V}_i) <: \sigma(\mathsf{T}_i)$ $(1 \le i \le n)$.

Define σ' as follows: $\sigma'(Y) = \sigma(K)$ if $Y \notin \{X_1, \ldots, X_n\}$ and $Y \in dom(\sigma)$; $\sigma(X_i) = \sigma(\mathsf{V})$ $(1 \le i \le n)$. It is clear that $\sigma' \setminus \{X_1, \ldots, X_n\} = \sigma$. To complete the proof for CT-Receive we must show that $(B, \sigma', \mathsf{Unit})$ is a solution for $(\Gamma, (x?x_1, \ldots, x_n), \mathsf{Unit}, \{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, X_1 <: \mathsf{T}_1, \ldots, X_n <: \mathsf{T}_n\})$, which requires σ' to satisfy $\{\mathsf{V} = \mathsf{chan}\{X_1, \ldots, X_n\}, X_1 <: \mathsf{T}_1, \ldots, X_n <: \mathsf{T}_n\}$. We have $\sigma'(\mathsf{V}) = \sigma'(\mathsf{chan}\{\mathsf{V}_1, \ldots, \mathsf{V}_n\}) = \mathsf{chan}\{\sigma'(\mathsf{V}_1), \ldots, \sigma'(\mathsf{V}_n)\} =$

$\mathsf{chan}\{\sigma(\mathsf{V}_1), \ldots, \sigma(\mathsf{V}_n)\}$ (since none of $X_1, \ldots, X_n$ appear in any of $\mathsf{V}_1, \ldots, \mathsf{V}_n$) $= \mathsf{chan}\{\sigma'(X_1), \ldots, \sigma'(X_n)\}$ (by definition of σ') $= \sigma'(\mathsf{chan}\{X_1, \ldots, X_n\}$. Finally, for $1 \leq i \leq n$, since none of $X_1, \ldots, X_n$ appear in T_i, $\sigma'(\mathsf{T}_i) = \sigma(\mathsf{T}_i$. We have $\sigma'(X_i) = \sigma(\mathsf{V}_i)$, and $B \vdash \sigma(\mathsf{V}_i) <: \sigma(\mathsf{T}_i) = \sigma'(\mathsf{T}_i)$. □

B.2 Proofs of results in Section 4.3

Lemma 4.1 Suppose $\Gamma \vdash S : \mathsf{U} \mid_\chi \mathcal{C}$. If $\mathsf{T}_1 <: \mathsf{T}_2 \in \mathcal{C}$ then at least one of $\mathsf{T}_1, \mathsf{T}_2$ is *not* a type variable.

Proof. The result follows from the fact that for any $PC_{partial}$ expression e, if $\Gamma \vdash \mathsf{e} : \mathsf{U}' \mid_{\chi'} \mathcal{C}'$ then U' is not a type variable. To see this, observe that all typing rules for expressions in Fig. 12 either assign a type from $\mathcal{A}^*$ (bool, Unit, or $\mathsf{T}_1 \vee \mathsf{T}_2$, where $\mathsf{T}_1, \mathsf{T}_2 \in \mathcal{A}^*$), or pass through a type computed by other rules. Exceptions to this are "source" rules CT-Bool-Lit, CT-Num-Lit and CT-Var. The first two of these rules generate base types. Rule CT-Var deduces the type for a variable reference x from the environment Γ. An entry in Γ comes from the rule CT-Decl, which adds $x : \mathsf{T}$ to Γ based on the type specified for x at declaration. This type is restricted to the syntax of Fig. 11, and thus cannot be a type variable.

Every rule in Fig. 12 which generates subtyping constraints produces constraints with one of three forms: $\mathsf{T}_1 <: \mathsf{T}_2$, $\mathsf{T}_1 <: X$, or $X <: \mathsf{T}_2$. In each case $\mathsf{T}_1, \mathsf{T}_2$ are types generated by the constraint typing relation for expressions, and thus are not type variables by the above argument. □

Lemma 4.2 Let $\mathcal{C}$ be a set of constraints and let B_0 be a bounds function such that $FV(\mathcal{C}) \subseteq dom(B_0)$. Then $unify(\mathcal{C}, B_0)$ terminates, either by failing or by returning a result of the form (B, σ).

Proof. The argument is the same as in Pierce [26, Theorem 22.4.5]. Given a set of constraints $\mathcal{C}$, let m be the number of distinct free type variables in $\mathcal{C}$ and let n be the total size of the types in $\mathcal{C}$. Then each recursive call of *unify* moves strictly downwards in the lexicographic order on (m, n). □

We prove Theorem 4.3, 20, via a series of technical lemmas.

Lemma B.1. *If (B, σ) unifies $\{X = \mathsf{T}\} \cup \mathcal{C}$ then (B, σ) unifies $[X \mapsto \mathsf{T}']\mathcal{C}$, where T' is either $\mu Y.F(Y)$ if T is of the form $F(X)$ (for some fresh type variable Y) or T otherwise.*

Proof. By induction on the size of $\mathcal{C}$. The base case, $\mathcal{C} = \emptyset$, is trivial because $[X \mapsto \mathsf{T}']\emptyset = \emptyset$.

If $\mathcal{C} = \{\mathsf{U} <: \mathsf{U}'\} \cup \mathcal{C}'$ then $[X \mapsto \mathsf{T}']\mathcal{C} = \{[X \mapsto \mathsf{T}']\mathsf{U} <: [X \mapsto \mathsf{T}']\mathsf{U}'\} \cup [X \mapsto \mathsf{T}']\mathcal{C}'$. Since (B, σ) unifies $\{X = \mathsf{T}\} \cup \mathcal{C}$ and $\mathcal{C}' \subset \mathcal{C}$ we have that (B, σ) unifies $\{X = \mathsf{T}\} \cup \mathcal{C}'$ and so, by the induction hypothesis, (B, σ) unifies $[X \mapsto \mathsf{T}']\mathcal{C}'$. It remains to show that (B, σ) unifies $[X \mapsto \mathsf{T}']\mathsf{U} <: [X \mapsto \mathsf{T}']\mathsf{U}'$, *i.e.* $B \vdash \sigma([X \mapsto \mathsf{T}']\mathsf{U}) <: \sigma([X \mapsto \mathsf{T}']\mathsf{U}')$. Because (B, σ) unifies $X = \mathsf{T}$, we have $\sigma(X) = \sigma(\mathsf{T}')$ (if T is of the form $F(X)$ then this requires equivalence of a recursive type and its unfolding) and therefore $\sigma([X \mapsto \mathsf{T}']\mathsf{U}) = \sigma(\mathsf{U})$, similarly $\sigma([X \mapsto \mathsf{T}']\mathsf{U}') = \sigma(\mathsf{U}')$. Since (B, σ) unifies $\mathsf{U} <: \mathsf{U}'$ we have $B \vdash \sigma(\mathsf{U}) <: \sigma(\mathsf{U}')$. This completes the argument.

If $\mathcal{C} = \{\mathsf{U} = \mathsf{U}'\} \cup \mathcal{C}'$ then $[X \mapsto \mathsf{T}']\mathcal{C} = \{[X \mapsto \mathsf{T}']\mathsf{U} = [X \mapsto \mathsf{T}']\mathsf{U}'\} \cup [X \mapsto \mathsf{T}']\mathcal{C}'$ and we require $\sigma([X \mapsto \mathsf{T}']\mathsf{U}) = \sigma([X \mapsto \mathsf{T}']\mathsf{U}')$. This follows from $\sigma(X) = \sigma(\mathsf{T}')$ as before. □

Lemma B.2. *If $(B, \sigma) \sqsubseteq (B', \sigma')$ and $\sigma'(X) = Y$ and $B(X) = (L_X, U_X)$ and $B'(Y) = (L_Y, U_Y)$ then $L_X <: L_Y <: U_Y <: U_X$.*

Proof. Let θ be an instance of (B', σ') such that $\theta(Y) = U_Y$. There exists γ such that $\theta = \gamma \circ \sigma'$, so $\theta(X) = \gamma(\sigma'(X)) = \gamma(Y)$ by hypothesis. Because (B', σ') is a bounded substitution and $\sigma'(X) = Y$, $Y \notin dom(\sigma')$, and so $\theta(Y) = \gamma(Y)$. Therefore $\theta(X) = \theta(Y)$. We have that θ is also an instance of (B, σ) and so $\theta(X) <: U_X$, *i.e.* $\theta(Y) <: U_X$, *i.e.* $U_Y <: U_X$ by choice of θ. Similarly by taking $\theta(Y) = L_Y$ we obtain $L_X <: L_Y$. We also have $L_Y <: U_Y$ by definition. The result follows. □

Lemma B.3. *If $(B, \sigma) \sqsubseteq (B', \sigma')$ and $B(X) = (L_X, U_X)$ and $B'(X) = (L'_X, U'_X)$ then $L_X <: L'_X <: U'_X <: U_X$.*

Proof. Similar to the proof of Lemma B.2: consider an instance θ of (B', σ') such that $\theta(X)$ is either L'_X or U'_X, and use the fact that θ is also an instance of (B, σ). □

Lemma B.4. *Let $L, U \in \mathcal{A}^*$, T a type which is not a type variable, and σ a substitution. Suppose $L <: \mathsf{T} <: U$. Then $L <: \sigma(\mathsf{T}) <: U$.*

Proof. If $\mathsf{T} \in \mathcal{A}^*$ then $\sigma(\mathsf{T}) = \mathsf{T}$ and the result follows. Otherwise, T is either a channel type or a product type. The only types in $\mathcal{A}^*$ to which T is comparable via the subtyping relation are therefore Bot and Top, so $L <: \mathsf{T} <: U$ forces $L = \mathsf{Bot}$ and $U = \mathsf{Top}$, from which we have $L <: \sigma(\mathsf{T}) <: U$ as required. □

Lemma B.5. *Let $\mathcal{C}$ be a set of constraints and let B_0 be a bounds function such that $FV(\mathcal{C}) \subseteq dom(B_0)$. Suppose that $unify(\mathcal{C}, B_0) = (B, \sigma)$. Then:*

1. *$dom(\sigma) \subseteq FV(\mathcal{C})$.*
2. *$dom(B) \cup dom(\sigma) = dom(B_0)$.*
3. *(B, σ) is a bounded substitution.*
4. *For every $X \in dom(B_0)$, $B \vdash L_X <: \sigma(X) <: U_X$ where $B_0(X) = (L_X, U_X)$.*
5. *(B, σ) is a unifier for $\mathcal{C}$.*

Proof. Each part is proved by induction on the number of recursive calls resulting from the initial call $\mathit{unify}(\mathcal{C}, B_0)$. The assumption that $\mathit{unify}(\mathcal{C}, B_0) = (B, \sigma)$ implies that $\mathit{unify}(\mathcal{C}, B_0)$ does not fail, enabling us to assume various conditions checked by the algorithm. Parts (1) and (2) are straightforward. The proof of part (3) uses (1) to show that when unifying $[Y \mapsto X]\mathcal{C}$, Y is not in the domain of the resulting substitution.

For part (4) some cases require more reasoning. If $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and so $(B, \sigma) = \mathit{unify}(\mathcal{C}', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \wedge \mathsf{T}_2)])$. Consider a type variable Z. If $Z \neq X$ then the induction hypothesis gives $B \vdash L_Z <: \sigma(Z) <: U_Z$ where $B_0(Z) = (L_Z, U_Z)$, as required for the conclusion. If $Z = X$ then the induction hypothesis gives $B \vdash L_X <: \sigma(X) <: U_X \wedge \mathsf{T}_2$, and the conclusion follows because $U_X \wedge \mathsf{T}_2 <: U_X$. The case where $\mathcal{C} = \{\mathsf{T}_1 <: X\} \cup \mathcal{C}'$ is symmetric. If $\mathcal{C} = \{X = Y\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$, and so $(B, \sigma) = (B', \sigma' \circ [Y \mapsto X])$ where $(B', \sigma') = \mathit{unify}([Y \mapsto X]\mathcal{C}', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \vee L_Y, U_X \wedge U_Y)])$. Consider a type variable Z. If $Z \neq X$ and $Z \neq Y$ then the induction hypothesis gives $B \vdash L_Z <: \sigma'(Z) <: U_Z$ where $B_0(Z) = (L_Z, U_Z)$, as required for the conclusion because $\sigma(Z) = \sigma'(Z)$. If $Z = X$ then the induction hypothesis gives $B \vdash L_X \vee L_Y <: \sigma'(X) <: U_X \wedge U_Y$, and the conclusion follows because $\sigma(X) = \sigma'(X)$ and $L_X <: L_X \vee L_Y$ and $U_X \wedge U_Y <: U_X$. If $Z = Y$ then similar reasoning concludes the argument, noting that $\sigma(Y) = \sigma'(X)$. If $\mathcal{C} = \{X = \mathsf{T}_2\} \cup \mathcal{C}'$ then the proof is straightforward induction.

For part (5) we again show the more interesting cases. If $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and so $(B, \sigma) = \mathit{unify}(\mathcal{C}', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \wedge \mathsf{T}_2)])$. By the induction hypothesis, (B, σ) unifies $\mathcal{C}'$. By (4), $B \vdash L_X <: \sigma(X) <: U_X \wedge \mathsf{T}_2$. Hence $B \vdash \sigma(X) <: \mathsf{T}_2$, and so (B, σ) unifies $\mathcal{C}$. The case where $\mathcal{C} = \{\mathsf{T}_1 <: X\} \cup \mathcal{C}'$ is symmetric. If $\mathcal{C} = \{X = Y\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$, and so $(B, \sigma) = (B', \sigma' \circ [Y \mapsto X])$ where $(B', \sigma') = \mathit{unify}([Y \mapsto X]\mathcal{C}', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \vee L_Y, U_X \wedge U_Y)])$. By the induction hypothesis (B', σ') unifies $[X \mapsto Y]\mathcal{C}'$ and therefore it is easy to see that (B, σ) unifies $\mathcal{C}'$; also it clearly unifies $X = Y$. If $\mathcal{C} = \{X = \mathsf{T}_2\} \cup \mathcal{C}'$ or $\mathcal{C} = \{\mathsf{T}_1 = X\} \cup \mathcal{C}'$ then similar reasoning applies. □

Lemma B.6. *Let $\mathcal{C}$ be a set of constraints and let B_0 be a bounds function such that $FV(\mathcal{C}) \subseteq dom(B_0)$. Suppose that (B, σ) is a unifier for $\mathcal{C}$ and $(B_0, [\,]) \sqsubseteq (B, \sigma)$. Then $\mathit{unify}(\mathcal{C}, B_0)$ does not fail.*

Proof. By induction on the number of recursive calls in $\mathit{unify}(\mathcal{C}, B_0)$. The base case is trivial.

If $\mathcal{C} = \{\mathsf{T}_1 <: \mathsf{T}_2\} \cup \mathcal{C}'$ and $\mathsf{T}_1, \mathsf{T}_2 \in \mathcal{A}$ then because (B, σ) unifies $\mathcal{C}$ we have $B \vdash \sigma(\mathsf{T}_1) <: \sigma(\mathsf{T}_2)$. Because $\mathsf{T}_1 \in \mathcal{A}$, $\sigma(\mathsf{T}_1) = \mathsf{T}_1$, and similarly for T_2. So $\mathsf{T}_1 <: \mathsf{T}_2$, and $\mathit{unify}(\mathcal{C}, B_0)$ does not fail.

If $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ and $\mathsf{T}_2 \in \mathcal{A}$ then let $(L_X, U_X) = B_0(X)$. Because (B, σ) unifies $\mathcal{C}$ we have $B \vdash \sigma(X) <: \mathsf{T}_2$. Now we consider whether or not $X \in dom(\sigma)$. If $X \notin dom(\sigma)$ then $B \vdash X <: \mathsf{T}_2$. Let $(L'_X, U'_X) = B(X)$, and then $U'_X <: \mathsf{T}_2$. Because $(B_0, [\,]) \sqsubseteq (B, \sigma)$, Lemma B.3 gives $L_X <: L'_X <: U'_X <: U_X$. Therefore $L_X <: U_X \wedge \mathsf{T}_2 \neq \mathsf{Bot}$ and so $\mathit{unify}(\mathcal{C}, B_0)$ does not fail. If $X \in dom(\sigma)$ then there are two further sub-cases. If $\sigma(X) = Y$ then let $(L_Y, U_Y) = B(Y)$ (because (B, σ) is a bounded substitution, $Y \notin dom(\sigma)$ and so $Y \in dom(B)$ by Lemma B.5). We have $B \vdash Y <: \mathsf{T}_2$, so $U_Y <: \mathsf{T}_2$. By Lemma B.2, $L_X <: L_Y <: U_Y <: U_X$ and so $L_X <: U_X \wedge \mathsf{T}_2 \neq \mathsf{Bot}$ as before. Finally, if $\sigma(X)$ is not a type variable then for a suitable γ, define $\theta = \gamma \circ \sigma$ so that θ is an instance of (B, σ). Because $(B_0, [\,]) \sqsubseteq (B, \sigma)$, θ is also an instance of $(B_0, [\,])$. Therefore $L_X <: \theta(X) <: U_X$. If $\sigma(X) \in \mathcal{A}$ then $\theta(X) = \sigma(X)$. If $\sigma(X) \notin \mathcal{A}$ then $\sigma(X)$ has the same subtyping relationships as $\theta(X)$ because its top-level type constructor is invariant for subtyping. In either case, we have $L_X <: \sigma(X) <: U_X$ and so $L_X <: \sigma(X) <: U_X \wedge \mathsf{T}_2 \neq \mathsf{Bot}$ as required.

If $\mathcal{C} = \{\mathsf{T}_1 <: X\} \cup \mathcal{C}'$ and $\mathsf{T}_1 \in \mathcal{A}$ then the reasoning is symmetrical to the previous case.

If $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ and $\mathsf{T}_2 \notin \mathcal{A}$ then *unify* replaces the subtyping constraint with $X = \mathsf{T}_2$, which is handled by the argument below; a similar argument applies when $\mathcal{C} = \{\mathsf{T}_1 = X\} \cup \mathcal{C}'$.

If $\mathcal{C} = \{X = Y\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$. Because (B, σ) unifies $X = Y$, we have $\sigma(X) = \sigma(Y)$. We now consider three cases. If $\sigma(X) = Y$ (and so $Y \notin dom(\sigma)$) then let $(L'_Y, U'_Y) = B(Y)$. By Lemma B.2, $L_X <: L'_Y <: U'_Y <: U_X$. By Lemma B.3, $L_Y <: L'_Y <: U'_Y <: U_Y$. Therefore $\mathsf{Top} \neq L_X \vee L_Y <: L'_Y <: U'_Y <: U_X \wedge U_Y \neq \mathsf{Bot}$ and $\mathit{unify}(\mathcal{C}, B_0)$ does not fail. If $\sigma(Y) = X$ then symmetrical reasoning applies. Finally, suppose that $X, Y \in dom(\sigma)$ and $\sigma(X) = \sigma(Y)$. There are two further sub-cases. If $\sigma(X) = \sigma(Y) = Z$, a type variable, then let $(L_Z, U_Z) = B(Z)$. Lemma B.2 gives $L_X <: L_Z <: U_Z <: U_X$ and $L_Y <: L_Z <: U_Z <: U_Y$, which is sufficient by similar reasoning to the previous case. If $\sigma(X) = \sigma(Y) = \mathsf{T}$, not a type variable, then for a suitable γ define $\theta = \gamma \circ \sigma$ so that θ is an instance of (B, σ). Because $(B_0, [\,]) \sqsubseteq (B, \sigma)$, θ is also an instance of $(B_0, [\,])$, so we have $L_X <: \theta(X) = \theta(Y) <: U_X$ and $L_Y <: \theta(X) = \theta(Y) <: U_Y$ which again is sufficient.

If $\mathcal{C} = \{X = \mathsf{T}_2\} \cup \mathcal{C}'$, where T_2 is not a type variable, then let $(L_X, U_X) = B_0(X)$. Since (B, σ) unifies $X = \mathsf{T}_2$ we have $\sigma(X) = \sigma(\mathsf{T}_2)$. Since T_2 is not a type variable this implies that $X \in dom(\sigma)$. For a suitable γ define $\theta = \gamma \circ \sigma$ so that θ is an instance of (B, σ). Because $(B_0, [\,]) \sqsubseteq (B, \sigma)$, θ is also an instance of $(B_0, [\,])$, so we have $L_X <: \theta(X) <: U_X$, *i.e.* $L_X <: \gamma(\sigma(X)) <: U_X$, *i.e.* $L_X <: \gamma(\sigma(\mathsf{T}_2)) <: U_X$, *i.e.* $L_X <: (\gamma \circ \sigma)(\mathsf{T}_2) <: U_X$. Applying Lemma B.4 we have $L_X <: \mathsf{T}_2 <: U_X$, which is what is needed for *unify* to succeed.

If $\mathcal{C} = \{\mathsf{T}_1 = X\} \cup \mathcal{C}'$, where T_1 is not a type variable, then the reasoning is symmetrical to the previous case. □

Lemma B.7. *Let $\mathcal{C}$ be a set of constraints and let B_0 be a bounds function such that $FV(\mathcal{C}) \subseteq dom(B_0)$. Suppose that (B, σ) is a unifier for $\mathcal{C}$ and $(B_0, [\,]) \sqsubseteq (B, \sigma)$. Then $\mathit{unify}(\mathcal{C}, B_0) \sqsubseteq (B, \sigma)$.*

Proof. Again the proof is by induction on the number of recursive calls in $\mathit{unify}(\mathcal{C}, B_0)$. We show the non-trivial cases.

If $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ and $\mathsf{T}_2 \in \mathcal{A}$ then let $(L_X, U_X) = B_0(X)$. We have $\mathit{unify}(\mathcal{C}, B_0) = \mathit{unify}(\mathcal{C}', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \wedge \mathsf{T}_2)]) = (B', \sigma')$ say. By hypothesis, (B, σ) unifies $\mathcal{C}'$. To use the induction hypothesis, which will complete the argument, we need $(B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \wedge \mathsf{T}_2)], [\,]) \sqsubseteq (B, \sigma)$. Let θ be an instance of (B, σ). So $\theta = \gamma \circ \sigma$ for some γ. To show that θ is an instance of $(B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \wedge \mathsf{T}_2)], [\,])$ we need $L_X <: \theta(X) <: U_X \wedge \mathsf{T}_2$. We already have $L_X <: \theta(X) <: U_X$. Also, because (B, σ) unifies $X <: \mathsf{T}_2$, $B \vdash \sigma(X) <: \mathsf{T}_2$. If $\sigma(X)$ is not a type variable then it must be a base type with $\sigma(X) <: \mathsf{T}_2$, and then $\theta(X) = \sigma(X)$ and we are done. If $\sigma(X) = Y$ then $Y \notin \mathit{dom}(\sigma)$ and $B \vdash Y <: \mathsf{T}_2$. Let $(L_Y, U_Y) = B(Y)$. Then $U_Y <: \mathsf{T}_2$. Now $\theta(X) = \theta(Y) = \gamma(Y)$, and $L_Y <: \theta(Y) <: U_Y$ because θ is an instance of (B, σ). Therefore $\theta(X) <: U_Y <: \mathsf{T}_2$. As we already have $L_X <: \theta(X) <: U_X$, we conclude that $L_X <: \theta(X) <: U_X \wedge \mathsf{T}_2$.

If $\mathcal{C} = \{\mathsf{T}_1 <: X\} \cup \mathcal{C}'$ and $\mathsf{T}_1 \in \mathcal{A}$ the result follows similarly.

If $\mathcal{C} = \{X = Y\} \cup \mathcal{C}'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$. We have $\mathit{unify}(\mathcal{C}, B_0) = (B', \sigma' \circ [Y \mapsto X])$ where $(B', \sigma') = \mathit{unify}([Y \mapsto X]\mathcal{C}', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \vee L_Y, U_X \wedge U_Y)])$. We need to show that $(B', \sigma' \circ [Y \mapsto X]) \sqsubseteq (B, \sigma)$. By Lemma B.1, (B, σ) unifies $[Y \mapsto X]\mathcal{C}'$. To use the induction hypothesis we need $(B_0 \setminus \{X, Y\} + [X \mapsto (L_X \vee L_Y, U_X \wedge U_Y)], [\,]) \sqsubseteq (B, \sigma)$. Let θ be an instance of (B, σ). Then $\theta = \gamma \circ \sigma$ for some γ. We need $L_X \vee L_Y <: \theta(X) <: U_X \wedge U_Y$. θ is also an instance of $(B_0, [\,])$, so $L_X <: \theta(X) <: U_X$ and $L_Y <: \theta(Y) <: U_Y$. But σ unifies $X = Y$, so $\sigma(X) = \sigma(Y)$ and so $\theta(X) = \theta(Y)$. Therefore $L_X \vee L_Y <: \theta(X) = \theta(Y) <: U_X \wedge U_Y$. Now, the induction hypothesis gives $(B', \sigma') \sqsubseteq (B, \sigma)$. Let ϕ be an instance of (B, σ); it is therefore also an instance of (B', σ'). We need to show that it is an instance of $(B', \sigma' \circ [Y \mapsto X])$. The condition on bounds follows directly from the fact that ϕ is an instance of (B', σ'). We also need the existence of γ such that $\phi = \gamma \circ \sigma' \circ [Y \mapsto X]$. Because ϕ is an instance of (B', σ'), there exists δ such that $\phi = \delta \circ \sigma'$. We will show that $\phi \circ [X \mapsto Y] = \phi$ and then take $\gamma = \delta$. Because ϕ is an instance of (B, σ), there exists η such that $\phi = \eta \circ \sigma$ and so, because $\sigma(X) = \sigma(Y)$, we have $\phi(X) = \phi(Y)$. Now consider the action of $\phi \circ [X \mapsto Y]$ on any type variable Z. If $Z \neq X$ then $(\phi \circ [X \mapsto Y])(Z) = \phi(Z)$. And $(\phi \circ [X \mapsto Y])(X) = \phi(Y) = \phi(X)$. Therefore $\phi \circ [X \mapsto Y] = \phi$, as required.

If $\mathcal{C} = \{X = \mathsf{T}_2\} \cup \mathcal{C}'$ and T_2 is not a type variable, then the reasoning is similar, and this reasoning also applies if $\mathcal{C} = \{X <: \mathsf{T}_2\} \cup \mathcal{C}'$ and $\mathsf{T}_2 \notin \mathcal{A}$, since *unify* replaces the subtyping constraint with $X = \mathsf{T}_2$; a similar argument applies when $\mathcal{C} = \{\mathsf{T}_1 = X\} \cup \mathcal{C}'$. □

Lemmas B.5–B.7 prove that the *unify* function computes the most general unifier for a set of constraints, when instantiated with a bounds function which assigns trivial bounds (Bot, Top) to every free type variable occurring in the constraints.

Theorem 4.3 Let $\mathcal{C}$ be a set of constraints and let B_0 be the bounds function defined by $\mathit{dom}(B_0) = \mathit{FV}(\mathcal{C})$ and $\forall X \in \mathit{dom}(B_0).B_0(X) = (\mathsf{Bot}, \mathsf{Top})$. If there exists a unifier for $\mathcal{C}$ then $\mathit{unify}(\mathcal{C}, B_0)$ is a unifier for $\mathcal{C}$, and for every unifier (B, σ) for $\mathcal{C}$, $\mathit{unify}(\mathcal{C}, B_0) \sqsubseteq (B, \sigma)$. That is, $\mathit{unify}(\mathcal{C}, B_0)$ is the most general unifier for $\mathcal{C}$.

Proof. Directly from Lemmas B.5–B.7. □

References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers—Principles, Techniques and Tools, Addison-Wesley, 1986.
[2] Alexander Aiken, Edward L. Wimmers, Type inclusion constraints and type inference, in: Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, Proceedings, ACM Press, 1993, pp. 31–41.
[3] Michael Baldamus, Jochen Schröder-Babo, p2b: A translation utility for linking Promela and symbolic model checking (tool paper), in: Matthew B. Dwyer (Ed.), Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, Proceedings, in: Lecture Notes in Computer Science, vol. 2057, Springer, 2001, pp. 183–191.
[4] Muffy Calder, Alice Miller, Feature interaction detection by pairwise analysis of LTL properties — a case study, Form. Methods Syst. Des. 28 (3) (2006) 213–261.
[5] Jeffrey Considine, Efficient hash-consing of recursive types, Technical Report 2000-006, Boston University, Computer Science, 2000.
[6] Marco Daniele, Paola Renditore, Roberto Manione, Timed simulation of distributed systems — from PROMELA to PROMELA+, in: Proceedings of the First International SPIN Workshop (SPIN'95): Complementary Material, 1995. Published online: http://spinroot.com/spin/Workshops/ws95/.
[7] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, Sophia Drossopolou, Session types for object-oriented languages, in: Dave Thomas (Ed.), ECOOP 2006 — Object-Oriented Programming, 20th European Conference, Nantes, France, Proceedings, in: LNCS, vol. 4067, Springer, 2006, pp. 328–352.
[8] Alastair F. Donaldson, Simon J. Gay, Etch: An enhanced type checking tool for Promela, in: Patrice Godefroid (Ed.), Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, Proceedings, in: Lecture Notes in Computer Science, vol. 3639, Springer, 2005, pp. 266–271.
[9] Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Type inference for recursively constrained types and its application to oop, Electron. Notes Theor. Comput. Sci. 1 (1995).
[10] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, Steven Levi, Language support for fast and reliable message-based communication in Singularity OS, in: Yolande Berbers, Willy Zwaenepoel (Eds.), Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, ACM Press, 2006, pp. 177–190.
[11] Christian Haack, Joe B. Wells, Type error slicing in implicitly typed higher-order languages, Sci. Comput. Programming 50 (1–3) (2004) 189–224.
[12] Klaus Havelund, Michael R. Lowry, John Penix, Formal analysis of a space-craft controller using SPIN, IEEE Trans. Software Eng. 27 (8) (2001) 749–765.
[13] Bastiaan J. Heeren, Top Quality Type Error Messages, Ph.D. Thesis, Universiteit Utrecht, The Netherlands, September 2005.
[14] Fritz Henglein, Jakob Rehof, The complexity of subtype entailment for simple types, in: 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, Proceedings, IEEE Computer Society Press LICS, 1997, pp. 352–361.
[15] J. Roger Hindley, The principal type-scheme of an object in combinatory logic, Trans. Amer. Math. Soc. 146 (1969) 29–60.
[16] Gerard J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
[17] Gerard J. Holzmann, Margaret H. Smith, Automating software feature verification, Bell Labs Tech. J. 5 (2) (2000) 72–87.
[18] Kohei Honda, Types for dyadic interaction, in: Eike Best (Ed.), CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, Proceedings, in: Lecture Notes in Computer Science, vol. 715, Springer, 1993, pp. 509–523.
[19] Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett, 2006.
[20] Laurent Mauborgne, Improving the representation of infinite trees to deal with sets of trees, in: Gert Smolka (Ed.), Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, Proceedings, in: Lecture Notes in Computer Science, vol. 1782, Springer, 2000, pp. 275–289.

[21] Robin Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.
[22] Robin Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), MIT Press, 1997.
[23] Martin Odersky, Martin Sulzmann, Martin Wehr, Type inference with constrained types, TAPOS 5 (1) (1999) 35–55.
[24] Fredrik Orava, Joachim Parrow, An algebraic verification of a mobile network, Formal Asp. Comput. 4 (6) (1992) 497–543.
[25] Simon L. Peyton Jones (Ed.), Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003.
[26] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002.
[27] François Pottier, A framework for type inference with subtyping, in: Proceedings of the International Conference on Functional Programming, ACM Press, 1998, pp. 228–238.
[28] François Pottier, Simplifying subtyping constraints: A theory, Inform. and Comput. 170 (2) (2001) 153–183.
[29] Vincent Rahli, J.B. Wells, Fairouz Kamareddine, Challenges of a type error slicer for the sml language, Technical Report HW-MACS-TR-0071, Heriot-Watt university, 2009.
[30] J. Alan Robinson, Computational logic: the unification computation, Mach. Intelligence 6 (1971) 63–72.
[31] Davide Sangiorgi, David Walker, The π-calculus: a Theory of Mobile Processes, Cambridge University Press, 2001.
[32] Francis Schneider, Steve M. Easterbrook, John R. Callahan, Gerard J. Holzmann, Validating requirements for fault tolerant systems using model checking, in: 3rd International Conference on Requirements Engineering (ICRE '98), Colorado Springs, CO, USA, Proceedings, IEEE Computer Society, 1998, pp. 4–13.
[33] Jerzy Tiuryn, Subtype inequalities, in: Proceedings of the IEEE Symposium on Logic in Computer Science, IEEE Press, 1992, pp. 308–315.
[34] Valery Trifonov, Scott F. Smith, Subtyping constrained types, in: Radhia Cousot, David A. Schmidt (Eds.), Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, Proceedings, in: Lecture Notes in Computer Science, vol. 1145, Springer, 1996, pp. 349–365.
[35] Bruce W. Watson, A taxonomy of finite automata minimization algorithms, Computing Science Report 93/44, Department of Computing Science, Eindhoven University of Technology, 1993.
[36] Andrew K. Wright, Matthias Felleisen, A syntactic approach to type soundness, Inform. and Comput. 115 (1) (1994) 38–94.