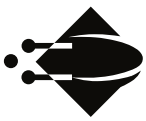
Formal Verification of Distributed Programs using Session Types and Coq

Morten Fangel Jensen

Advisors: Jesper Bengtson and Fabrizio Montesi
Submitted: June 2014



IT University
of Copenhagen

Abstract

This thesis investigates one possible way of proving functional correctness of distributed programs by integrating Session Types and Separation Logic. We introduce a language that has Session Type style typing judgements inside its Hoare triples and Separation Logic predicates inside its Session Type protocols. The language – which builds on an existing Java-like language built by Bengtson *et al.* [3] – is implemented in Coq, which has also been used to prove properties and theorems of the language semantics. The extended language has the new *send*, *recv* and *start* primitives with which programs with multiple processes that communicate using message-passing governed by Session Type protocols can be constructed. Using Separation Logic predicates instead of types to describe the transferred data, receivers are allowed to assume properties proven by the sending party.

As a case-study, we introduce a version of distributed merge sort implemented in our language and show its correctness using decorated programs on paper. The results show that our language has the ability to reason about the functional correctness of recursive, distributed programs using Hoare-triples and Separation Logic predicates in a way that is similar to how non-distributed programs are verified.

Contents

Chapter 1

Introduction

It is difficult to implement programs correctly. Knuth once pointed out that it took 16 years from when the first version of binary search was published until the first correct version was published [12]. Other famous bugs, like those of Therac-25 [13] and Ariane 5 [14], show that it does not get easier when the size of the program increases. Adding in the fact that many modern programs are distributed, opens up the possibility to further problems. Incorrect use of sending/receiving actions can lead to deadlocks when both sides of a communication are stuck waiting for each other indefinitely.

To ensure that non-distributed programs are bug-free, there has been a lot of focus and research on the topic of formal verification using Hoare logic [8] and Separation Logic [16]. Using interactive proof assistants, it is possible to guarantee full functional correctness of some programs. This type of reasoning is currently rather labour intensive. Even worse, it is only possible to reason about a subset of features and programming languages. But for some programs it is possible. A key aspect of formal verifications is the predicates with which requirements for correctness are defined. These predicates state properties about data contained within programs, and these can be used to prove properties of programs as a whole.

To help developers avoid common problems with distributed programs, researchers turned to Session Types [9]. By introducing a typing system inspired by Linear Logic [7], it is possible to guarantee protocol compliance within communicating programs. The protocols enforceable by Session Types are currently limited to describing the types of the transferred data. Knowing only the type of the transferred data makes

it impossible to reason about the functional correctness of programs relying on this data as any properties the sender has proven will not be conveyed to the receiver.

To be able to prove properties about transferred data, the programming language model could include the ability to transfer predicates that describe the data exchanged between sender and receiver. Without predicates describing the transferred data, the functional correctness of any program receiving data can not be proven.

Problem Statement

Current frameworks for reasoning about functional correctness do not support predicates that describe data exchanged between senders and receivers and thus do not support reasoning about correctness of distributed programs.

By integrating Session Types into the Hoare logic used to prove functional correctness, protocol compliance can be proven for programs. Furthermore, if we let Session Types protocols contain predicates instead of only types, full functional correctness can be proven for distributed programs. Instead of ensuring that data is of a certain type before sending, the sending program must first prove the predicate that is asserted for the data being transferred. Reversely, programs receiving data are allowed to assume that the predicate describing the data holds.

There is an inherent mismatch between Session Types and Hoare logic because their direction of reasoning is slightly different. In both cases, the proof obligation is to prove that a command performs a certain modification of the starting state. In Session Types, this could be that the command actually performs the next action required by the protocol. Where they differ is that with Session Types you are allowed to assume that the continuation of the protocol will be handled by the remaining part of the program. With Hoare logic you need to prove that the actions of the commands result in a state where the given postcondition then must hold. However, in this thesis we will present a model that allows for the construction of Hoare triples that can verify protocol compliance on a step-by-step basis. On each sending action, part of the protocol compliance verification would be to show that the predicate describing the data being sent holds.

Hypothesis

It is possible to integrate Session Types in Hoare logic in such a way that compliance to protocols containing predicates can be verified step-by-step.

In this thesis we investigate a way of integrating Session Types and Hoare logic and demonstrate how full functional correctness for distributed programs is proven.

Thesis Statement

Integrating Session Types and Hoare logic provides a good foundation for the verification of distributed programs.

We will test our hypothesis by developing a language model with Hoare logic rules that uses Session Type based predicates to reason about network communication. We will then prove the full functional correctness of distributed programs to show the validity of our model.

1.1 Contributions

This thesis contains three main contributions.

Firstly, we developed a language model of distributed programs. We did this by extending the language developed by Bengtson *et al.* in [3]. The extension consists of adding three new primitives, *send*, *recv* and *start*, and extending the memory model with a notion of communication channels. Importantly, we also show that the extension retains all properties of the existing model.

Secondly, we created a set of Hoare-triples for reasoning about the behavior of the communication primitives with which we have extended our language. This allows verifying distributed programs implemented in our language.

Lastly, we tested our model using a case study. We evaluated how to represent and verify an implementation of a distributed merge sort as well as other simpler programs.

The first two contributions have been implemented and mechanized in Coq. The case study is done as a series of decorated programs presented in this thesis.

The contents of this thesis have been written entirely by the author. However, the advisors were heavily involved with what the semantics should be defined as. The language implementation, as well as all mechanized proofs of properties and axioms, was either pre-existing or produced by the author for this thesis.

Appendix A contains information on how to access the Coq source-code as well as a list of admitted lemmas.

Chapter 2

Preview

To demonstrate how our logic can be used to verify programs involving network communication, we have chosen a naive distributed implementation of merge sort as our example. This naive implementation splits the list in two and then spawns two new instances of itself to handle the sorting of the sublists The source-code for the implementation in our Java-like language can be seen in Fig. 2.1.

For the purpose of this thesis, we have opted to leave out the sequential implementations of both the *split* and the *merge* functions, but we will later give a specification that any implementation must respect. This thesis is an extension of an existing Java-like language introduced by Bengtson *et al.* in [3], and this language already allows for verification of such functions. Instead we focus on the distributed aspect, which is where our contributions lie.

The existing Java-like language relies on Hoare logic, which was introduced by Hoare in [8] to prove the functional correctness of programs. Programs are proven on a line-by-line basis through a language-dependent set of Hoare-triples. These triples are normally written in the form $\{P\}c\{Q\}$, where P is a precondition, c is a command and Q is a postcondition. Hoare-triples only deal with partial correctness, and the triple reads "if P holds in the initial state, and if c terminates, then Q holds in the resulting state". It is important to note that if c does not terminate, Hoare logic offers no insight as to what happens.
The definition of triples is often extended with a notion of safety, which implies that the command does not lead to a faulty state. So a side-

```
protocol p :
      ?i, {List i xs}.
      !o, {List o ys∧
           Sorted_of xs ys}.
      ϵ

class MergeSort {
  method split(l) {
    ⋮
  }
  method merge(l1, l2) {
    ⋮
  }
  method sort(l) {
    x = start MergeSort::MS p;
    send x l;
    s = recv x;
    return s
  }
  method MS(x) {
    l = recv x;
    if l.length() ≤ 1 {
      send x l
    } else {
      t = MergeSort::split l;
      ll = t.fst;
      lr = t.snd;
      xl = start MergeSort::MS p;
      xr = start MergeSort::MS p;
      send xl ll;
      send xr lr;
      sl = recv xl;
      sr = recv xr;
      s = MergeSort::merge(sl, sr);
      send x s
    };
    return null
  }
}
```

Figure 2.1: Source code for our distributed merge sort example

condition is added to the triple that ensures that if P holds in the initial state, then c can not result in a faulty state.

There is a shared set of Hoare-triples that are applicable to most imperative programming languages. These cover primitives such as sequencing, if-else branches, loops, etc. For instance, we can define a sequencing rule such that we can reason about programs consisting of multiple commands that applies to most imperative languages:

$$\frac{\{P\}\, c_1\, \{Q\} \qquad \{Q\}\, c_2\, \{R\}}{\{P\}\, c_1; c_2\, \{R\}}\ \textsc{Rule-Seq}$$

To make reasoning about mutable data structures located on the heap feasible, Reynolds introduced Separation Logic in [16]. A key part to

Separation Logic is the separating conjunction – denoted by $*$ – which says that the two parts of the conjunction hold in disjoint parts of the heap [16, 17]. This allows us to create predicates of the form $\alpha * \beta$, which means that α and β must hold in separate parts of the heap. The rules for reasoning within Separation Logic are described in Fig. 2.2

$$\frac{}{P * Q \vdash Q * P} \qquad \frac{}{(P * Q) * R \vdash Q * (P * R)} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * R \vdash R} \qquad \frac{P \vdash Q}{P * R \vdash Q * R} \qquad \frac{}{P * \mathbf{true} \vdash P} \qquad \frac{}{P \vdash P * \mathbf{true}}$$

Figure 2.2: The rules our Intuitionistic Separation Logic must satisfy

The most basic Separation Logic predicate is the *pointsto* predicate, which is written in the form $v \mapsto v'$. In classical Separation Logic the predicate holds if the heap it is evaluated in is a singleton heap containing only one entry v', at the address v. If you were to use normal conjunctions, the assertion $x \mapsto v' \wedge y \mapsto v''$ would imply that $x = y \wedge v' = v''$ because both parts of the conjunction requires a singleton heap. Thus, the only way to construct a singleton heap such that both *pointsto*-predicates holds is if the predicates talk about the same element. In Intuitionistic Separation Logic there are no restrictions on what else the heap might contain. On the other hand, the assertion $v \mapsto v' * v \mapsto v''$ is contradicting because the heaps for each *pointsto* needs to be distinct but both needs to contain the address v.

As we mentioned earlier, we are able to reason about the functionality of the omitted methods *split* and *merge* using Separation Logic. But there is currently no way to reason about the primitives *send*, *recv* and *start*, which are the three additions to the language we propose.

To help reasoning about network communication, Honda *et al.* introduced the concept of Session Types in [9] which is inspired by Linear Logic, introduced by Girard in [7]. Session Types were later extended with subtypes by Gay & Hole in [6].

Session Types use the notion of protocols, which describe the network communication that needs to be performed on channels. In the

example program given in Fig. 2.1, we define such a protocol on the first line of the program. Having to explicitly define protocols is not uncommon within Session Type implementations and can also be seen in, e.g., Session-J as introduced by Hu *et al.* in [10], but in Section 6.1.1 we will discuss our reasons for requiring explicit protocol definitions.

The standard syntax of Session Types can be seen in Fig. 2.3. The inhabitants of the transferable types depend on the implementation of Session Types but are often defined to be the union of the implementation language's native types and S. As visible in our example program from Fig. 2.1, we use a slightly different syntax. Instead of denoting the data to transfer by way of its type(s) ,$[T_1, \ldots, T_n]$, we use a tuple consisting of a variable name and a predicate which describes the data.

$$
\begin{array}{lll}
S ::= \epsilon & & \textit{terminated session} \\
\quad | \quad ?[T_1, T_2, \ldots, T_n].S & & \textit{input} \\
\quad | \quad ![T_1, T_2, \ldots, T_n].S & & \textit{output} \\
\quad | \quad \&\langle l_1 : S_1, \ldots, l_n : S_n \rangle & & \textit{branch} \\
\quad | \quad \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle & & \textit{choice} \\
\quad | \quad \mu T.S & & \textit{recursive protocol} \\
\\
\quad T, T_1, \ldots, T_n \in \textit{transferable types} & &
\end{array}
$$

Figure 2.3: The standard syntax of Session Types [6, Fig. 1]

Having predicates in Session Type protocols allows our protocols to define more information about the transferred data. For instance, the protocol $?z, \{P\,z\}.T$ can be read as "receive a z where the assertion $P\,z$ holds, then continue as protocol T". The variable name acts as a placeholder for the value that is sent or received. So if the channel x has the Session Type $?z, \{P\,z\}.T$, after calling **recv** $x\,y$, we will know the predicate $P\,y$ because the placeholder z has been substituted for the variable we stored the value in: y.

In the method *sort* from our example, we need to start *MergeSort::MS* as a new process and setup a communication channel to it. To reason about the functionality of *start*, we again turn to Session Types and its definition of when two processes can communicate. In Session Types, two processes can communicate iff the two processes' channels have protocols that are each others dual. The intuition is that dual protocols

$$
\begin{aligned}
\overline{\epsilon} &= \epsilon \\
\overline{T} &= T \\
\overline{?[T_1, T_2, \ldots, T_n].S} &= ![T_1, T_2, \ldots, T_n].\overline{S} \\
\overline{![T_1, T_2, \ldots, T_n].S} &= ?[T_1, T_2, \ldots, T_n].\overline{S} \\
\overline{\&\langle l_1 : S_1, \ldots, l_n : S_n\rangle} &= \oplus\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\rangle \\
\overline{\oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle} &= \&\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n}\rangle \\
\overline{\mu T.S} &= \mu T.\overline{S}
\end{aligned}
$$

Figure 2.4: The dual of a Session Types protocol [6, Fig. 2]

are the opposite of each other, so when one party expects to send, the other expects to receive, etc. The rules for duality can be seen in Fig. 2.4.

In *start*, after we have started the method *MS* with the protocol p, the local channel x will have the protocol $\overline{p}$. The started method – *MS* – will receive a channel with the protocol p as its sole argument. A requirement imposed by *start* is that when the method terminates, all channels it received or started itself must be terminated.

If we look at *sort*, we see all three new primitives in use. First, *start* is used to create the merge sort process. Because the channel has the dual protocol of p, the first action is to *send* a list. Assuming that the argument passed to *sort* is a list, then proving the *List*-predicate from the protocol is trivial. The *List l xs*-predicate will be formally defined later but holds when the object pointed to by l contains the logical list xs. By calling *recv*, we will not only receive a new list, but also the predicate *Sorted_of xs ys*[1] informing us that the contents of the received list, ys, is the sorted version of the list we sent, xs. Thus, when *sort* returns the received list, we are able to prove that the method as a whole returns the sorted version of the list given as an argument.

The actual implementation of merge sort is in the method *MS*. The method is started as a new process with the protocol p by *start* and then immediately performs the receive-action to retrieve the list it is supposed to sort. If the list is empty or a singleton list, it is by definition

[1]The *Sorted_of*-predicate will be formally defined in Def. 3.8.

sorted, in which case we simply return the list again. The predicate describing the data we are sending is $\exists ys,\ List\ l\ ys \wedge Sorted\_of\ xs\ ys$, which holds if we send back a list representing the sorted version of xs. As xs contains zero or one element(s), xs itself ensures that the predicate holds. The send also terminates the protocol and fulfills the requirement from *start* that all channels have finished their protocols.

If the list contains multiple elements, we need to merge sort the list. This is done by splitting the list into a left and right part, which gets stored in the variables *ll* and *lr*. The method then starts off two *MS*-subprocesses and sends the lists to these. We know from the protocol that the lists we receive back from the subprocesses will be sorted, so we can simply merge the two received lists into the final sorted list. This merged and sorted list we can now send back. Once we have completed the final send, we know that both the protocol of the initial channel, as well as the protocols of the two subprocess channels, have terminated, which fulfills the termination-requirement from *start*.

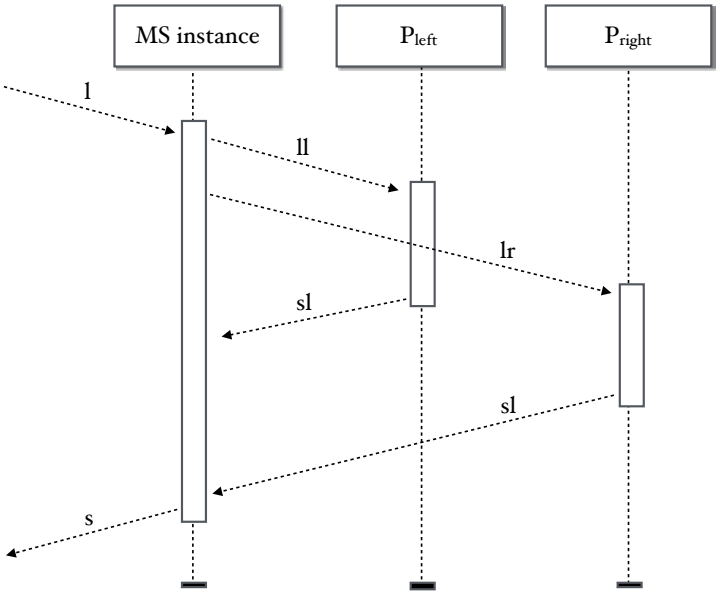


Figure 2.5: A graph showing the communication flow in the distributed merge sort example

The flow of data between an instance of *MS* and its two subprocesses, denoted P_{left} and P_{right}, can be seen in Fig. 2.5. It is clear from this diagram that the two subprocesses can perform their sorting in parallel, which allows us to call our implementation distributed.

2.1 Method specifications

When verifying the functional correctness of programs, the usual approach is to create mathematical specifications that each method of the program must respect. We define functional correctness to mean that the function satisfies its specifications, which means that the verification is only as useful as the given specifications. Once every function satisfies its specification, full functional correctness of the program has been proven.

We write specifications for method on the form $C::m(\bar{a}) \mapsto \{P\}\_\{r.\, Q\}$. The exact definition will be given later in Def. 3.17, but it can be thought of as meaning "If you know P, then after you call $C::m$ with the arguments $\bar{a}$, you will known Q". The postcondition can rely on the return-value, named r.

The method *split* has to take a list and split it into two lists. This can be represented by the following method specification:

$$\begin{aligned} Split\_spec \triangleq \forall xs, \text{MergeSort::split}(\mathbf{l}) \mapsto \{List\, \mathbf{l}\, xs\}\_ \\ \{\mathbf{r}.\, \exists r_1\, r_2\, xs_1\, xs_2,\, \mathbf{r}.fst \hookrightarrow r_1 * List\, r_1\, xs_1 \wedge |xs_1| \geq 1 * \\ \mathbf{r}.snd \hookrightarrow r_2 * List\, r_2\, xs_2 \wedge |xs_2| \geq 1 \wedge xs = xs_1 +\!\!+\, xs_2 \end{aligned}$$

The predicate $v.f \hookrightarrow v'$ is a slight variation of the *pointsto*-predicate, which will be formally defined in Def. 3.6. It holds iff the entry v' is located in the heap address $v.f$.
The specification can be read as indicating that it takes a list representing xs and produces a pair of lists that represents xs_1 and xs_2, where the two lists combine to form xs. It is never specified that we would prefer the lists to be split into two roughly equal length lists. But as long as neither of the lists are the empty list, **nil**, then our merge sort implementation will finish.

The specification for the *merge*-method can be given as the following:

$$\begin{aligned} Merge\_spec \triangleq \forall xs_1\, xs_2\, ys_1\, ys_2, \text{MergeSort::merge}(\mathbf{l1}, \mathbf{l2}) \mapsto \{List\, \mathbf{l1}\, ys_1 * \\ List\, \mathbf{l2}\, ys_2 \wedge Sorted\_of\, xs_1\, ys_1 \wedge \\ Sorted\_of\, xs_2\, ys_2\}\_ \\ \{\mathbf{r}.\, \exists ys,\, List\, \mathbf{r}\, ys \wedge Sorted\_of\, (xs_1 +\!\!+\, xs_2)\, ys\} \end{aligned}$$

It can be read as producing a sorted list containing $xs_1 +\!\!+\, xs_2$, if given two sorted lists where one contains xs_1 and the other xs_2.

Our distributed merge sort needs to receive a list, and then return the sorted list. So the protocol, which we call MSP, for communicating with an instance of *MergeSort::MS* is:

$$MSP \triangleq ?\mathbf{i}, \{List\ \mathbf{i}\ xs\}.!\mathbf{o}, \{List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys\}.\epsilon$$

With the protocol established, we can define the precondition which must hold before the method specification for *MergeSort::MS* can be proven. This precondition is $x : MSP$, where predicates of the type $c : T$, holds when the channel c has the Session Type T. We also know that every channel must be terminated when *MergeSort::MS* finishes, which we can express using the predicate $all\_ST : \epsilon$.

A requirement to calling *start* is that the protocol exists and is equal to the Session Type that the method expects. So the method specification needs the assumption $p : MSP$ to ensure that we can start the MS subprocesses and communicate with them according to the MSP-protocol. Thus the specification is

$$MS\_spec \triangleq p : MSP \vdash \text{MergeSort::MS}(x) \mapsto \{x : MSP\}\_\{r.\, all\_ST : \epsilon\}$$

The last specification we need is the specification that any users of our merge sort will rely on: The specification for the method *sort*. It needs to take a list as its argument and return a sorted equivalent. To do this, the protocol p must exists and the method MS must be follow and complete the protocol MSP.

The specification for *sort* thus becomes the following:

$$Sort\_spec \triangleq p : MSP \vdash \text{MergeSort::sort}(l) \mapsto \{List\ l\ xs\}\_ \\ \{r.\, \exists ys,\ List\ r\ ys \wedge Sorted\_of\ xs\ ys\}$$

Using our language and its Hoare-triples, it is possible to show that the implementation of distributed merge sort shown in Fig. 2.1 conforms to the above four specifications. As we believe that the specifications are valid, it implies that, if the specifications are followed, our distributed merge sort implementation has full functional correctness.

Theorem 2.1 (Distributed Merge Sort)**.** *The implementation of distributed merge sort given in Fig. 2.1 conforms to the specification* $Merge\_spec \wedge Split\_spec \wedge Sort\_spec \wedge MS\_spec$.

The next chapter will formalize our language, and, in Chapter 4, we show decorated programs that shows the validity of the claim in the theorem above.

Chapter 3

Formalization

So far we have been somewhat vague in what all the constructions and predicates mean. This section serves to make precise all aspects of our language which involve defining its operational and axiomatic semantics. Our case-study relies on the Hoare-triples and predicates defined in this section to verify the correctness of our example programs.

3.1 Language Syntax

The syntax of our language is almost identical to the original syntax described by Bengtson *et al.* in [3], except for our addition of the primitives *start*, *send* and *recv* which we introduced in the previous section. The entire syntax of our language can be seen in Fig. 3.1.
Between the publication of [3] and the beginning of this thesis, the existing model was modified to use a deep embedding of expressions instead of the shallow embedding described in the paper. So the published formalization of the existing model has the grammar for e missing.

The language is weakly typed, and values are defined as a union of integers, booleans, object-pointers, array-pointers, channel identifiers and *null*. There is a casting mechanism in place which uses a default value if the casting could not be performed.

As visible from Fig. 3.1, the protocol map is not currently represented in the syntax of the program. Instead we rely on predicates specified on the left-hand side of our method specifications. The predicates we use to reason about the protocol map will be given later. It would, however, be straightforward to change the definition of $\mathcal{P}$ such that it includes the protocol map. The reason why the protocol map is left out

$$
\begin{aligned}
\mathcal{P} &::= \mathcal{C}^* \\
\mathcal{C} &::= \textbf{class}\, C\, \{\, f^*\, (m(\bar{x})\{c; \textbf{return}\, e\})^*\, \} \\
c &::= x = \textbf{alloc}\, C \mid x = e \mid x = y.f \mid x.f = e \mid x = y.m(\bar{e}) \\
&\quad \mid\ x = C{::}m(\bar{e}) \mid \textbf{skip} \mid c_1; c_2 \mid \textbf{if}\, e\, \{c_1\}\, \textbf{else}\, \{c_2\} \\
&\quad \mid\ \textbf{while}\, e\, \{c\} \mid \textbf{assert}\, e \\
&\quad \mid\ x = \textbf{start}\, C{::}m\, p \mid \textbf{send}\, x\, y \mid y = \textbf{recv}\, x \\
e &::= v \mid x \mid e + e \mid e - e \mid e * e \mid e \wedge e \\
&\quad \mid\ e \vee e \mid \neg e \mid e < e \mid e = e
\end{aligned}
$$

$$
x, y \in \textit{variable-names}, f \in \textit{field-names}, v \in \textit{values},
$$
$$
C \in \textit{class-names}, p \in \textit{protocol-identifiers}
$$

Figure 3.1: The syntax of our language

of the program definition is that we hope to later evolve the language such that the protocol map is no longer required.

3.2 Memory Model

The memory model of our language is almost identical to that of the model we are extending. Both the notion of the stack and heap come directly from the existing model, but, because the model has evolved since it was described in [3], we have both definitions reproduced here.

Firstly the stack is – as one would expect – a total mapping of variable names to values. Since the mapping is total, we require that a default value exists, such that a value can always be returned. We use the special value *null* in case a variable name is not represented on the stack.

Definition 3.1 (stack)**.** *The stack is a function from variable-names to values.*

$$
\mathit{stack} \triangleq \mathit{var} \to \mathit{val}
$$

Along with the stack, Bengtson *et al.* introduce the notion of *open* types. It is defined as $\mathit{open}\, T \triangleq \mathit{stack} \to T$. Intuitively it can be thought of as a type that requires a stack to be computed. As an example, expressions can be defined as $\mathit{expr} \triangleq \mathit{open}\, \mathit{val}$ – or in other words: expressions that use program variables can be evaluated to a value in the presence of a stack.

This generic definition of *open* types allows for an equally generic definition of *substitutions*. Any *open* types can have their program variables substituted for values. This substitution is implemented by wrapping the *open* type with the substitutions, such that any lookup of the substituted variable returns the substituted value instead. We use the notation $e\{^{v}/_{a}\}$ to mean the expression e, where the variable a is replaced with the value v. There is also a notion of *truncating substitutions* with the notation $e\{^{\overline{v}}/_{a}\}$ which beside substituting the variable a, also replaces any other variable with the value *null*.

The definition of the heap in our memory model is taken directly from the original model. However, the model has evolved since it was first described in [3]. In the original paper, the heap could only contain objects. The authors of the paper later extended it to also contain arrays. The way the heap is now modeled is that objects are kept in one part of the heap while arrays are kept in another disjoint part of the heap. In other words, the heap is a pair containing an object-heap and an array-heap.

Definition 3.2 (heap)**.** *The heap is a pair of partial functions. The first from object-pointers and fields to values and the second from array-pointers and offsets to values.*

$$\begin{aligned} heap_{ptr} &\triangleq (ptr * field) \rightharpoonup val \\ heap_{arr} &\triangleq (arrptr * nat) \rightharpoonup val \\ heap &\triangleq (heap_{ptr} * heap_{arr}) \end{aligned}$$

$$ptr \in \textit{object-pointers}, field \in \textit{field-names}, arrptr \in \textit{array-pointers}$$

If h has the type $heap$, we write h_{ptr} and h_{arr} to refer to the individual sections of the heap.

We use the operator $\circ$ to compose two heaps into a new bigger heap iff the two heaps are compatible. Two heaps are compatible, written $h_1 \# h_2$, if the domains of the two heaps are disjoint. We also have a notion of a heap being the subheap of another. A heap s is said to be a subheap of another heap h, iff there exists a compatible heap which combines with s such that the composed heaps equals h: $h \sqsupseteq s \triangleq \exists h', s \circ h' = h$.

The only addition to the memory model of the original language we propose is a Session Type map. This map contains the Session Type of all active channels.

Definition 3.3 (Session type map)**.** *The Session Type map, denoted* S*, is a partial mapping of channel identifiers to Session Types*

$$\mathrm{S} \triangleq \mathit{stptr} \rightharpoonup \mathit{ST}$$

$$\mathit{stptr} \in \mathit{channel\text{-}identifiers}$$

Note that ST refers to our definition of Session Types which can be found in Section 3.3.1.

The protocol map described earlier is equivalent, although it maps from *protocol-identifiers* instead of *channel-identifiers*. We denote the protocol map with P.

Notation Both the object and array sections of the heap, along with the Session Types and protocol map, are all implemented using maps. We will use the following notations for interacting with these maps: If you have a map m, a key k and a value v, then $k \in m$ tests membership, $m(k) = v$ check that the value stored at the key k has the value v. Lastly we use the notation $m(k \leftarrow v)$ to assign the value v to the key k. We use the notation $[k \leftarrow v]$ for the singleton map containing only the key k with the value v.

3.3 Assertion Logic

Our model has two types of logic: The assertion logic and the specification logic. In this section we will describe the assertion logic and then in Section 3.5 will we introduce the specification logic.

The purpose of the assertion logic is to describe predicates that capture properties that hold in a given state. The predicates used in the pre- and postconditions of Hoare-triples are thus instances of our assertion logic. The assertions can be thought of as functions that are passed a state – represented by variables of the types described in the previous section – and returns `Prop`, the natural type for a predicate in Coq.
The existing language contained the two assertion logics *asn* and *sasn*, and we have added the additional logics *pasn* and *psasn* such that the state of the Session Type map can be assessed with predicates in our assertion logic.

Definition 3.4 (Existing Assertion Logic)**.** *The assertion logic, sasn, defined by Bengtson et al. is a function from stack, programs, natural numbers and heap to Prop. The asn-logic is downwards closed on the natural number and upwards closed on the program and heap.*

$$\begin{aligned} asn &\triangleq \mathcal{P}^{\uparrow} \to \mathbb{N}^{\downarrow} \to heap^{\uparrow} \to \texttt{Prop} \\ sasn &\triangleq open\, asn \end{aligned}$$

The up- and downwards closures imply that the result will not change if the the input is respectively increased or decreased. So the upwards closure on heaps implies that any predicate that holds in a heap h will also hold in bigger heap h', if $h' \sqsupseteq h$. The downwards closure on the natural number – which functions as a step-index – implies that given any lower number, the predicate will still hold. The upwards closure on programs implies that the predicate will still hold even if additional class definitions are introduced.

The framework which the existing language is built on top of – Charge!, as introduced in [2] – is able to automatically infer that the above assertion logic is an Intuitionistic Separation Logic, using type-classes inspired by the works of Dockins *et al.* in [5]. This means that $P * Q$ implies that P and Q holds in the same stack, program and step-index but in separate parts of the heap.

The aim of this thesis is to support both assertion logic predicates inside Session Types, as well as Session Type judgements inside our assertion logic. Fulfilling both of these requirements would lead to a circular dependency between the Session Types definition and the definition of our assertion logic. To avoid this, we have structured our assertion logic in two levels. The first level is *sasn*, which can reason about the stack and heap, which is what we embed within Session Types. The second level will be introduced after we have defined our Session Types, and it allows reasoning about the active Session Type channels.

3.3.1 Having Predicates Inside Session Types

Part of this thesis' contributions is a way to integrate predicates inside Session Type protocols such that properties of the transferred data can be better specified. With the *sasn* assertion logic defined, we can formally define our version of Session Types.

Currently our language only supports sending and receiving actions, and thus does not support branching, choice or recursion. In Section 6.2 we discuss how more of Session Types could be added to our language. The syntax we use for Session Types is

$$ST ::= !v, \{P\}.ST \mid ?v, \{P\}.ST \mid \epsilon$$

$$v \in \textit{variable-names}, P \in \textit{sasn}$$

where $!v, \{P\}$ denotes sending the predicate P, $?v, \{P\}$ denotes receiving P and ϵ denotes the terminated protocol.

As mentioned earlier, the variable acts as a placeholder or entry-point into the given predicate. If the channel x has the protocol $!v\{v = 5\}.\epsilon$ and the command $\textit{send}\, x\, y$ is run, the predicate to prove would be $y = 5$ because the variable we are sending, y, has been substituted for the placeholder v. The proof-obligation and substitution is formally defined as part of the axiomatic semantics presented in Section 3.6.

Because ST is not an *open* type, the standard definition of substitutions does not apply. As we want the ability to perform a substitution on the tail of a Session Type, we create the following recursive definition of substitution on Session Types:

$$\begin{aligned}
\epsilon\{^{v}/_{a}\} &= \epsilon \\
(!z, \{P\}.ST)\{^{v}/_{a}\} &= !z, \{P\{^{v}/_{a}\}\}.ST\{^{v}/_{a}\} \\
(?z, \{P\}.ST)\{^{v}/_{a}\} &= ?z, \{P\{^{v}/_{a}\}\}.ST\{^{v}/_{a}\}
\end{aligned}$$

Our definition assumes that each placeholder has a district name. The substitution would apply to any later predicates that re-uses a placeholder name, which will likely render those predicates useless. An alternative definition would stop the recursive substitution if the placeholder has the same name as the variable being replaced.

With our definition of Session Types in place, we can define our extended assertion logic which is able to reason on the active communication channels.

Definition 3.5 (Extended Assertion Logic)**.** *Our extended assertion logic, psasn, is a function from stack, protocol maps, Session Type map, programs,*

natural numbers and heap to Prop. The logic is downwards closed on the natural number and upwards closed on the program and heap.

$$\begin{aligned} \mathit{pasn} &\triangleq \mathrm{P} \to \mathrm{S} \to \mathit{asn} \\ \mathit{psasn} &\triangleq \mathit{open}\ \mathit{pasn} \end{aligned}$$

The reason why *pspec* was not defined using *sasn* is that by having the assertion be an *open* type, we can continue to use the standard substitution and lifting functionality of the existing language.

Our definition of *pasn* and *psasn* assertions also ensures that it is possible to embed *asn* or *sasn* assertions within them. The possibility to embed the assertions means that all the specifications and triples from the existing language can be easily reused.

One immediate downside to this two-layered assertion logic is that Session Types can not contain predicates that reason about other Session Types. But as we will discuss in Section 6.2, this does not hinder the ability to later extend our model with delegation, although new primitives would be needed in order to delegate channels.

Charge! – the framework used to construct the assertion logics – currently does not support building a Separation Logic which is separate over two parameters of the logic. In our extended assertion logic, $P * Q$ should imply that P and Q hold in the same stack, program, protocol map and step-index, but in separate heaps and Session Type maps. So like the existing assertion logic, our extended logic is only separate on the heap-argument. This limitation impacts our ability to prove the frame-rule – which will be discussed in Section 3.6 – for *psasn*-assertions. As we will later discuss, this means the mechanized proof of our frame rule has been left admitted. However, once *psasn* can be constructed to be separate on both the heap and the Session Type map, proving the frame-rule will be trivial.

3.3.2 Predicates in our Assertion Logic

In order to write method specifications for the four methods in our merge sort example, we needed a few predicates defined in our assertion logic. The first was the *pointsto* predicate. Because the assertion logic used is upwards closed on heaps, all of our assertions are intuitionistic

assertions. Thus our *pointsto*-predicate does not imply singleton heaps like the classic definition does.

Definition 3.6 (pointsto)**.** $v.f \hookrightarrow v'$[1] *(pronounced "$v.f$ points to v'"), holds if* $[(v, f) \mapsto v']$ *is a subset of the current heap.*

$$v.f \hookrightarrow v' \triangleq h_{ptr} \sqsupseteq [(v, f) \mapsto v']$$

Where h_{ptr} *is the pointer-section of the heap in which the assertion is evaluated.*

The second assertion we rely on is the *List*-predicate, which we have adapted from the *List_rep*-predicate given by Mehnert in [15].

Definition 3.7 (List)**.** *The predicate "List α i" holds if the data-structure in α represents the logical list i.*

$$Node\_list\ \alpha\ xs \triangleq \begin{cases} \alpha = \textbf{\textit{null}} & \text{if } xs = \textbf{\textit{nil}} \\ \exists \alpha', \alpha.val \hookrightarrow x * \alpha.next \hookrightarrow \alpha' * \\ \qquad Node\_list\ \alpha'\ xs' & \text{if } xs = x :: xs' \end{cases}$$

$$List\ \alpha\ i \triangleq \exists h, \alpha.head \hookrightarrow h * Node\_list\ h\ i$$

We also introduce the pure assertion $Sorted\_of$, which describes a relationship between two logical variables.

Definition 3.8 (Sorted_of)**.** *For all list of values xs and ys, $Sorted\_of\ xs\ ys$ holds if xs and ys contains the same values and ys is sorted.*

All of the predicates described so far could have been implemented in the assertion logic of the existing language, *sasn*, because they only verify properties related to the contents of the stack or heap.

The next two predicates describe the Session Types of ongoing communication channels and would have been impossible to implement in the earlier model.

The first predicate shares its form with how typing judgments commonly look and it also has the same intuition behind it: Is a given item of a particular type? In our case, does a channel have a given Session Type associated with it?

[1] We use $v.f \hookrightarrow v'$ instead of $v.f \mapsto v'$ because our assertions are upwards closed on heaps. Reynolds uses the definition $v.f \hookrightarrow v' \triangleq v.f \mapsto v' * \textbf{true}$ [17] which intuitively has the same meaning as the upwards closure of heaps.

Definition 3.9 (channel-has-type)**.** *The predicate $x : T$, which implies that the channel x is of the Session Type T, holds iff the entry for x in the Session Type map contains T.*

$$x : T \triangleq \mathrm{S}(x) = T \qquad x \in \mathit{val}\text{, type cast as a channel-identifier}$$

Where S *is the Session Type map in which the assertion is evaluated.*

For the specification of methods that are startable, we need a predicate for when every single protocol has terminated. We do this by introducing the *all_ST*-predicate which holds iff every Session Type in the Session Type map has a given protocol – e.g. ϵ.

Definition 3.10 (all_ST)**.** *The predicate $all\_ST : T$ holds iff every entry in the Session Type map has contains T.*

$$all\_ST : T \triangleq \forall c,\, c \in \mathrm{S} \rightarrow \mathrm{S}(c) = T$$

3.4 Operational Semantics

The operational semantics define the actions performed by the commands that make up the language. The operational semantics of the existing model is a big-step semantics, which means that each command is defined by a relation containing the starting state, the command, the resulting state and how many computational steps it took to get there. The commonly used syntax is $c, \sigma \rightsquigarrow^n \sigma'$, which can be read as: Command c, starting in state σ results in state σ' in n computational steps. A special state is the state **err**, which is used to denote errors – e.g., a memory access violation.

As defined in Section 3.2, the state used in the original model consisted of $\mathit{stack} \times \mathit{heap}$. In our extended model, the state also contains the ongoing Session Types. In addition to the starting state, the command is also given its context, which consists of the program definition as well as the protocol map. So Definition 2 through 4 from [3] is redefined to include this addition and thus becomes the following[2]:

Definition 3.11 (pre-command)**.** *A pre-command $\hat{c}$ relates an initial state to either a terminal state or the special **err** state:*

$$\mathit{precmd} \triangleq \mathbb{P}(\mathcal{P} \times \mathrm{P} \times \mathit{stack} \times \mathit{heap} \times S \times \mathbb{N} \times ((\mathit{stack} \times \mathit{heap} \times \mathrm{S}) \uplus \boldsymbol{\mathit{err}}))$$

[2]Safe never had a separate definition in [3] but was given inside the definition of the frame property

We write $(\hat{c}, \mathcal{P}, \mathrm{P}, s, h, t) \rightsquigarrow^n \sigma$ *to denote that* $(\mathcal{P}, \mathrm{P}, s, h, t, n, \sigma) \in \hat{c}$*. We consider* $\mathcal{P}$ *and* P *implicit and thus also write* $(\hat{c}, s, h, t) \rightsquigarrow^n \sigma$ *to denote the same.*

Definition 3.12 (Safe)**.** *A pre-command* $\hat{c}$ *is said to be safe if the initial state does not lead to the special* ***err*** *state:*

$$\mathit{safe}\,\hat{c}\,s\,h\,t\,n \triangleq (\hat{c}, s, h, t) \not\rightsquigarrow^n \textbf{\textit{err}}$$

Definition 3.13 (Frame property)**.** *A pre-command* $\hat{c}$ *has the frame property if* $(\hat{c}, s, h \circ h_{frame}, t \circ t_{frame}) \rightsquigarrow^n (s', h'_{big}, t'_{big})$ *and* $\mathit{safe}\,\hat{c}\,s\,h\,t\,n$ *implies that there exists* h' *and* t' *such that* $h'_{big} = h' \circ h_{frame}$*,* $t'_{big} = t' \circ t_{frame}$ *and* $(\hat{c}, s, h, t) \rightsquigarrow^n (s', h', t')$*.*

Using pre-commands and the frame property, we can define what it means to be a semantic command.

Definition 3.14 (Semantic command)**.** *A semantic command is a pre-command that satisfies the frame property and has no evaluation that takes zero computational steps.*

$$\mathit{semcmd} \triangleq \{\hat{c} \in \mathit{precmd} \mid \hat{c}\ \text{has the frame-property} \wedge \forall s, h, t, \sigma,\ (\hat{c}, s, h, t) \not\rightsquigarrow^0 \sigma\}$$

The existing semantic commands were modified to accommodate the addition of the Session Types to the state in a straight forward manner. Since none of the existing commands involved any network communication, they just need to pass the Session Types of the initial state along. The semantic commands inherited from the language by Bengtson *et al.* are

$$\textbf{id} \quad \textbf{seq}\,\hat{c_1}\,\hat{c_2} \quad \hat{c}_1 + \hat{c}_2 \quad \hat{c}^* \quad \textbf{assume}\,P \quad \textbf{assign}\,x\,e$$
$$\textbf{new}\,x\,C \quad \textbf{read}\,x\,y.f \quad \textbf{write}\,x.f\,e \quad \textbf{call}\,x\,\mathrm{C}{::}\mathrm{m}(\bar{e})\,\textbf{with}\,c\,\hat{c}$$

and the inductive constructors for the subset of commands used in this thesis can be seen in Fig. 3.2.

One semantic command that requires some explanation is the command for method calls. The special **with** $c\,\hat{c}$-section of the command represents the method body and its corresponding semantic command. This single semantic command is able to represent both static and dynamic methods since the object reference can be passed as an additional parameter to the method, similarly to what e.g. Python does.

$$\frac{}{(\mathbf{id}, s, h, t) \rightsquigarrow^1 (s, h, t)}\ \textsc{Id-OK} \qquad \frac{(\hat{c}_1, s, h, t) \rightsquigarrow^n \mathbf{err}}{(\mathbf{seq}\,\hat{c}_1\,\hat{c}_2, s, h, t) \rightsquigarrow^{n+1} \mathbf{err}}\ \textsc{Seq-FailL}$$

$$\frac{(\hat{c}_1, s, h, t) \rightsquigarrow^n (s', h', t') \qquad (\hat{c}_2, s', h', t') \rightsquigarrow^m \mathbf{err}}{(\mathbf{seq}\,\hat{c}_1\,\hat{c}_2, s, h, t) \rightsquigarrow^{n+m+1} \mathbf{err}}\ \textsc{Seq-FailR}$$

$$\frac{(\hat{c}_1, s, h, t) \rightsquigarrow^n (s', h', t') \qquad (\hat{c}_2, s', h', t') \rightsquigarrow^m (s'', h'', t'')}{(\mathbf{seq}\,\hat{c}_1\,\hat{c}_2, s, h, t) \rightsquigarrow^{n+m+1} (s'', h'', t'')}\ \textsc{Seq-OK}$$

$$\frac{(\hat{c}_1, s, h, t) \rightsquigarrow^n (s', h', t')}{(\hat{c}_1 + \hat{c}_2, s, h, t) \rightsquigarrow^{n+1} (s', h', t')}\ \text{+-OkL} \qquad \frac{(\hat{c}_1, s, h, t) \rightsquigarrow^n \mathbf{err}}{(\hat{c}_1 + \hat{c}_2, s, h, t) \rightsquigarrow^{n+1} \mathbf{err}}\ \text{+-FailL}$$

$$\frac{(\hat{c}_2, s, h, t) \rightsquigarrow^n (s', h', t')}{(\hat{c}_1 + \hat{c}_2, s, h, t) \rightsquigarrow^{n+1} (s', h', t')}\ \text{+-OkR} \qquad \frac{(\hat{c}_2, s, h, t) \rightsquigarrow^n \mathbf{err}}{(\hat{c}_1 + \hat{c}_2, s, h, t) \rightsquigarrow^{n+1} \mathbf{err}}\ \text{+-FailR}$$

$$\frac{P\,s}{(\mathbf{assume}\,P, s, h, t) \rightsquigarrow^1 (s, h, t)} \qquad \frac{eval\,e\,s = v}{(\mathbf{assign}\,x\,e, s, h, t) \rightsquigarrow^1 (s(x \to v), h, t)}$$

$$\textsc{Read-OK}\ \frac{s\,y = p \qquad h_{\text{ptr}}((p, f)) = v}{(\mathbf{read}\,x\,y.f, s, h, t) \rightsquigarrow^1 (s(x \to v), h, t)} \qquad \textsc{Read-Fail}\ \frac{s\,y = p \qquad h_{\text{ptr}}((p, f)) \neq v}{(\mathbf{read}\,x\,y.f, s, h, t) \rightsquigarrow^1 \mathbf{err}}$$

$$\textsc{Write-OK}\ \frac{s\,x = p \qquad (p, f) \in h_{\text{ptr}} \qquad eval\,e\,s = v}{(\mathbf{write}\,x.f\,e, s, h, t) \rightsquigarrow^1 (s, h((p, f) \to v), t)} \qquad \textsc{Write-Fail}\ \frac{s\,x = p \qquad (p, f) \notin h_{\text{ptr}}}{(\mathbf{write}\,x.f\,e, s, h, t) \rightsquigarrow^1 \mathbf{err}}$$

$$\textsc{Call-FailS}\ \frac{\forall r,\ \mathrm{C::m}(\overline{p})\{c; \mathbf{return}\,r\} \notin \mathcal{P}}{(\mathbf{call}\,x\,\mathrm{C::m}(\overline{e})\,\mathbf{with}\,c\,\hat{c}, s, h, t) \rightsquigarrow^1 \mathbf{err}} \qquad \textsc{Call-FailC}\ \frac{\mathrm{C::m}(\overline{p})\{c; \mathbf{return}\,r\} \in \mathcal{P} \quad |\overline{p}| \neq |\overline{e}|}{(\mathbf{call}\,x\,\mathrm{C::m}(\overline{e})\,\mathbf{with}\,c\,\hat{c}, s, h, t) \rightsquigarrow^1 \mathbf{err}}$$

$$\frac{\mathrm{C::m}(\overline{p})\{c; \mathbf{return}\,r\} \in \mathcal{P} \quad |\overline{p}| = |\overline{e}| \quad (\hat{c}, [s \leftarrow \overline{e}], h, t) \rightsquigarrow^n \mathbf{err}}{(\mathbf{call}\,x\,\mathrm{C::m}(\overline{e})\,\mathbf{with}\,c\,\hat{c}, s, h, t) \rightsquigarrow^{n+1} \mathbf{err}}\ \textsc{Call-FailC}$$

$$\frac{\mathrm{C::m}(\overline{p})\{c; \mathbf{return}\,r\} \in \mathcal{P} \quad |\overline{p}| = |\overline{e}| \quad (\hat{c}, [s \leftarrow \overline{e}], h, t) \rightsquigarrow^n (s', h', t')}{(\mathbf{call}\,x\,\mathrm{C::m}(\overline{e})\,\mathbf{with}\,c\,\hat{c}, s, h, t) \rightsquigarrow^1 (s(x \leftarrow s'\,r), h', t')}\ \textsc{Call-OK}$$

Figure 3.2: The constructors for the used subset of the inherited semantic commands

The semantics of our programming language commands are defined by relating them to semantic commands. We, like Bengtson *et al.*, will use the notation

$$c \sim_{\text{sem}} \hat{c}$$

for this relation. The semantic relation of the commands in our language can be seen in Fig. 3.3.

$$\frac{}{\textbf{skip} \sim_{\text{sem}} \textbf{id}}\ \textsc{Skip-Sem} \qquad \frac{c_1 \sim_{\text{sem}} \hat{c}_1 \qquad c_2 \sim_{\text{sem}} \hat{c}_2}{c_1; c_2 \sim_{\text{sem}} \textbf{seq}\, \hat{c}_1\, \hat{c}_2}\ \textsc{Seq-Sem}$$

$$\frac{c_1 \sim_{\text{sem}} \hat{c}_1 \qquad c_2 \sim_{\text{sem}} \hat{c}_2}{\textbf{if}\, e\, \{c_1\}\, \textbf{else}\, \{c_2\} \sim_{\text{sem}} (\textbf{seq}\, (\textbf{assume}\, e)\, \hat{c}_1) + (\textbf{seq}\, (\textbf{assume}\, \neg e)\, \hat{c}_2)}\ \textsc{If-Sem}$$

$$\textsc{Assign-Sem}\ \frac{}{x = e \sim_{\text{sem}} \textbf{assign}\, x\, e} \qquad \textsc{Read-Sem}\ \frac{}{x = y.f \sim_{\text{sem}} \textbf{read}\, x\, y.f} \qquad \textsc{Write-Sem}\ \frac{}{x.f = e \sim_{\text{sem}} \textbf{write}\, x.f\, e}$$

$$\frac{c \sim_{\text{sem}} \hat{c}}{x = \text{C::m}(\overline{e}) \sim_{\text{sem}} \textbf{call}\, x\, \text{C::m}(\overline{p})\, \textbf{with}\, c\, \hat{c}}\ \textsc{SCall-Sem}$$

$$\frac{c \sim_{\text{sem}} \hat{c} \qquad y : C}{x = \text{y::m}(\overline{e}) \sim_{\text{sem}} \textbf{call}\, x\, \text{C::m}(y, \overline{p})\, \textbf{with}\, c\, \hat{c}}\ \textsc{DCall-Sem}$$

Figure 3.3: The subset of utilized program commands related to their semantic commands

In order to specify the semantics of the three new language primitives, we must give a semantic command corresponding to how the primitives should behave. We must then show how these semantic commands relate to the primitives through the $\sim_{\text{sem}}$-relation. The next three sections correspond to the semantic command and relation for each of the new primitives.

3.4.1 Send

What intuitively happens when you ask to send a variable v is that all the data related to this variable is transferred. This is trivial in the case of scalar data types but less clear when it comes to data structures like

objects or arrays. In case the variable is a pointer to a data structure, we must transfer a subset of our heap that contains at least the object we desire to send. The Session Type of the channel we are communicating on must also be modified to reflect that the send has occurred.

Our model never actually transfers any data, but only identifies the value and heap that needs to be transferred. If our language was to be implemented, the semantics need to describe that the data is pushed to the recipient along the indicated channel.

For send, we define the pre-command

$$\textbf{send}\, x\, v$$

which sends the value of v and, in the case that v is a pointer to a data structure, a heap that contains at least this data structure. The pre-command is defined using the following three constructors

$$\frac{s\,x = c \qquad t(c) = !z, \{P\}.T \qquad \exists h', h \sqsupseteq h' \wedge P\,[z \leftarrow s\,v]\,h'\,n \qquad \forall h', \mathcal{D}_{sasn}\,P\,[z \leftarrow s\,v]\,h'\,n}{(\textbf{send}\, x\, v, s, h, t) \rightsquigarrow^{n+1} (s, h, t(c \leftarrow T))}\ \textsc{Send-OK}$$

$$\frac{s\,x = c \qquad t(c) = !z, \{P\}.T \qquad \neg P\,[z \leftarrow s\,v]\,h\,n}{(\textbf{send}\, x\, v, s, h, t) \rightsquigarrow^{n+1} \textbf{err}}\ \textsc{Send-FailP} \qquad \frac{s\,x = c \qquad c \notin t}{(\textbf{send}\, x\, v, s, h, t) \rightsquigarrow^{1} \textbf{err}}\ \textsc{Send-FailC}$$

where $c \in$ channel-identifiers

Common for all three constructors is that they identify the channel-identifier c by looking up the channel x on the stack. If the channel is not in the Session Type map t, the send will fail, which is what the Send-FailC-constructor implies. Otherwise, we assume that the Session Type of c will be the sending of the predicate P.

The problem for send is how to identify the subheap that needs to be transferred. We have chosen to rely on the predicate, as it must hold in the the transferred heap. Thus the success case finds a subset of the heap where P holds. The assumption responsible for finding the subheap is $\exists h', h \sqsupseteq h' \wedge P\,[z \leftarrow s\,v]\,h'\,n$. In case P does not hold in the entire heap to begin with, Send-FailC evaluate the send to the **err** state. This method for identifying the subheap might first seem weird and hard to implement in a compiler. However, a naive implementation

could just select the entire starting heap h. If our compiler only accepts verified programs along with the proof of correctness, the Hoare triple for sending contains exactly the proof that P holds in the heap h. If the compiler only accepts verified programs, the failure case will also never be used and can thus be ignored.

The downside to utilizing the predicate to identify which subheap to transfer is that it introduces a requirement of decidability, with the $\mathcal{D}_{\text{sasn}}$-predicate. What this decidability requirement implies is that the transferred predicate must either hold or not hold in a given state. Without this requirement it is impossible to prove that the command has the frame property. The decidability requirement will be formally introduced in Section 3.6.2 and discussed further in Section 6.1.2.

We show that the pre-command satisfies the frame property by performing a case-study on whether or not P holds in the unframed heap. Since no constructor defines an evaluation taking zero steps, the pre-command is a semantic command.

Theorem 3.1 (send_cmd)**.** *The pre-command* ***send*** *x v with the inductive constructors* Send-OK, Send-FailO *and* Send-FailC *is a semantic command.*

The relation between the program command and the semantic command is straightforward

$$\frac{}{\textbf{send}\ x\ y \sim_{\text{sem}} \textbf{send}\ x\ v}\ \textsc{Send-Sem}$$

3.4.2 Receive

For receiving, what you intuitively want to achieve is to assign a local program variable, v, to the value that was transferred. If the transferred value was a pointer to a data structure, you want the same data structure to be present in your own heap and v to be a pointer to the structure. Again, our model does not actually perform the communication, so any implementation would have to read these from the network.

We define the following pre-command, which receives the value rv and the heap rh from the sender on the channel x and stores the received value in v

$$\textbf{recv}\ x\ v$$

with the following constructors:

$$\frac{s\,x = c \qquad t(c) = ?z, \{P\}.T \qquad \forall n, P[z \leftarrow rv]\,rh\,n}{(\mathbf{recv}\,x\,v, s, h, t) \rightsquigarrow^1 (s(v \leftarrow rv),\ h \circ rh,\ t(c \leftarrow T))}\ \textsc{Recv-OK}$$

$$\frac{s\,x = c \qquad c \notin t}{(\mathbf{recv}\,x\,v, s, h, t) \rightsquigarrow^1 \mathbf{err}}\ \textsc{Recv-Fail}$$

where $c \in$ channel-identifiers

The requirement is that the predicate P holds in the received heap rh. The resulting state is the received value rv pushed to the stack, the heap extended by rh and the protocol advanced to the tail of the protocol. The idea is that we are allowed to introduce P because the sender proved that P held in the sent heap.

Our implementation of $\mathbf{recv}\,x\,v$ currently assumes that the current and received heaps are compatible. We realize this might not always be the case. Our model only requires that the predicate P holds in the received heap, so, if the semantics were to be implemented, every address in the received heap could be offset in such a way that it becomes compatible with the current heap. The received value as well as any pointers inside data-structures in the received heap would also need to be updated to reflect the offset in addresses.

Because $\mathbf{recv}\,x\,v$ does not rely on the initial heap, proving that the pre-command has the frame property is trivial. This leads us to the fact that the pre-semantic command is also a semantic command.

Theorem 3.2 (recv_cmd)**.** *The pre-command* ***recv*** *x v with the inductive constructor* Recv-OK *and* Recv-Fail *is a semantic command.*

The relation to the *recv* program primitive is equally straightforward in that it simply unifies the variables:

$$\frac{}{y = \mathbf{recv}\,x \sim_{\text{sem}} \mathbf{recv}\,x\,v}\ \textsc{Recv-Sem}$$

3.4.3 Start

The last primitive we need to define the semantics for is the *start*-command. We have chosen to model it on method calls, so the semantic

command looks like the **call**-command given in [3]. We define the pre-command

$$\textbf{start}\, x\, \mathrm{C{::}m}(a)\, p\, \textbf{with}\, b\, \hat{b}$$

which intuitively starts method m of class C with the communication-channel instantiated in the variable a. The communication channel is also assigned to the local program variable x. The command b is the method-body of m, and $\hat{b}$ is the corresponding semantic command.

In the success case we look up the protocol p from the protocol-map P to the Session Type T, find an unused channel-identifier c and check that the method $\mathrm{C{::}m}(a)$ exists in the program $\mathcal{P}$. The semantic command corresponding to the method-body is used to find the resulting state and iff this Session Type map contains only terminated channels, the *start*-command succeeds. The resulting state has c added to the stack and $\overline{T}$ added to the Session Type map.

The error cases cover either an ongoing channel – i.e. a channel that was not terminated by the started method (Start-FailO) – or errors in the started method – i.e. when the started method results in the **err**-state (Start-FailE).

$$\frac{\begin{array}{c}\mathrm{P}(p) = T \qquad c \notin t \qquad \mathrm{C{::}m}(a)\{b; \textbf{return}\, r\} \in \mathcal{P} \\ (\hat{b}, [a \leftarrow c], \textbf{emp}, [c \leftarrow T]) \rightsquigarrow^{n} (s', h', t') \qquad \forall T', t'(T') = \epsilon\end{array}}{(\textbf{start}\, x\, \mathrm{C{::}m}(a)\, p\, \textbf{with}\, b\, \hat{b}, s, h, t) \rightsquigarrow^{n+1} (s(x \leftarrow c), h, t(c \leftarrow \overline{T}))}\ \textsc{Start-OK}$$

$$\frac{\begin{array}{c}\mathrm{P}(p) = T \qquad c \notin t \qquad \mathrm{C{::}m}(a)\{b; \textbf{return}\, r\} \in \mathcal{P} \\ (\hat{b}, [a \leftarrow c], \textbf{emp}, [c \leftarrow T]) \rightsquigarrow^{n} \textbf{err}\end{array}}{(\textbf{start}\, x\, \mathrm{C{::}m}(a)\, p\, \textbf{with}\, b\, \hat{b}, s, h, t) \rightsquigarrow^{n+1} \textbf{err}}\ \textsc{Start-FailE}$$

$$\frac{\begin{array}{c}\mathrm{P}(p) = T \qquad c \notin t \qquad \mathrm{C{::}m}(a)\{b; \textbf{return}\, r\} \in \mathcal{P} \\ (\hat{b}, [a \leftarrow c], \textbf{emp}, [c \leftarrow T]) \rightsquigarrow^{n} (s', h', t') \\ \exists T', T' \in t' \wedge t'(T') \neq \epsilon\end{array}}{(\textbf{start}\, x\, \mathrm{C{::}m}(a)\, p\, \textbf{with}\, b\, \hat{b}, s, h, t) \rightsquigarrow^{n+1} \textbf{err}}\ \textsc{Start-FailO}$$

where $c \in$ channel-identifiers

Similarly to *recv*, *start* does not depend on the initial heap and thus it is trivial to prove that the frame property holds. This tells us that our pre-command is a semantic command.

Theorem 3.3 (start_cmd)**.** *The pre-command* ***start*** *x C::m*(a) *p* ***with*** *b* $\hat{b}$ *with the inductive constructors* Start-OK, Start-FailE *and* Start-FailO *is a semantic command.*

The relation between the program command *start* and the semantic command is similar to method calls. It requires that b has the semantic command $\hat{b}$.

$$\frac{b \sim_{\text{sem}} \hat{b}}{x = \textbf{start}\,\text{C::m}\,p \sim_{\text{sem}} \textbf{start}\,x\,\text{C::m}(a)\,p\,\textbf{with}\,b\,\hat{b}}\ \textsc{Start-Sem}$$

3.5 Specification Logic

The specification logic is – like the name implies – the logic in which program, class and method specifications are written. The specification logic that already exists from [3], *spec*, was based only on the program definition and a step index. We have added the protocol map to the specification logic, such that predicates in the specification logics can express requirements that certain protocols must be in the map.

Like with the assertion logic, predicates in our specification logic can be considered functions into `Prop`.

Definition 3.15 (Specification Logic)**.** *The extended specification logic, pspec, is a function from protocol maps, programs and natural numbers to Prop. The logic is downwards closed on the natural number and upwards closed on the program.*

$$\begin{aligned} spec &\triangleq \mathcal{P}^{\uparrow} \to \mathbb{N}^{\downarrow} \to \texttt{Prop} \\ pspec &\triangleq \text{P} \to spec \end{aligned}$$

A requirement of Bengtson *et al.* for their language was to have the specification logic embeddable within the assertion logic. This requirement explains why $\mathcal{P}$ is part of the definition of their assertion logic. To ensure that specifications continue to be embeddable, our extended assertion logic was also made to contain the protocol map P.

3.5.1 Predicates in our Specification Logic

The building-blocks for program verifications are the Hoare-triples that cover the behavior of the program commands. As mentioned earlier, the

form of these triples are $\{P\}c\{Q\}$, where P and Q are predicates in our assertion logic and c is a command. The intuition is that if P holds and c terminates, then Q holds in the resulting state.

Definition 3.16 (Hoare triple)**.** *The Hoare triple $\{P\}c\{Q\}$ holds if, given that P holds, the command is safe and if the command terminates, it does so in a state where Q holds.*

$$\begin{aligned}\{P\}c\{Q\} \triangleq \forall \hat{c}\, \mathcal{P}\, \mathrm{P}\, s\, h\, t\, n\, m,\ c \sim_{sem} \hat{c} \rightarrow P\, s\, \mathrm{P}\, t\, \mathcal{P}\, n\, h \rightarrow (\mathit{safe}\, \hat{c}\, s\, h\, t\, n \\ \wedge\ (\forall s'\, h'\, t',\ (\hat{c}, s, h, t) \rightsquigarrow^{m} (s', h', t') \rightarrow Q\, s', \mathrm{P}\, t'\, \mathcal{P}\, (n - m)\, h'))\end{aligned}$$

Using the Hoare-triple defined by the axiomatic semantics, it is possible to construct a triple that covers the entirety of a method-body. It is thus possible to describe the specification of a method by saying what Hoare-triple the method-body has.

Definition 3.17 (Method specification)**.** *A method $C::m(\bar{p})$ has the specification $C::m(\bar{p}) \mapsto \{P\}\_\{r.\, Q\}$ if the body c of the method conforms to the triple $\{P\}c\{Q\}$.*

$$\begin{aligned}C::m(\bar{p}) \mapsto \{P\}\_\{r.\, Q\} \triangleq C::m(\bar{p}')\{c; \mathbf{return}\ e\} \in \mathcal{P} \\ \wedge\ |\bar{p}| = |\bar{p}'| \wedge \{P\{\overline{\bar{p}/\bar{p}'}\}\}c\{Q\{\overline{e.\bar{p}/r.\bar{p}'}\}\}\end{aligned}$$

The definition can be read as ensuring that the method is defined in the program $\mathcal{P}$, and that the specification and definition agrees on the number of arguments. The Hoare-triple with the method-body c and the pre- and postconditions from the specification must then hold. The substitution serves to enable different parameter names between the method definition and its specification. The substitution is truncating, which means that any arguments not mentioned in the method definition will be replaced by *null*.

The only new specification logic predicate we have added is the predicate $p : T$ such that we can specify which protocols are present in the protocol map.

Definition 3.18 (protocol-has-type)**.** *A protocol p is said to be the Session Type T, iff the entry in the protocol map contains T.*

$$p : T \triangleq \mathrm{P}(p) = T \qquad p \in \textit{protocol-identifier}$$

Where P *is the protocol map in which the specification is evaluated.*

3.6 Axiomatic Semantics

To finalize the formalization of our language, we need to define the Hoare-triples that allow reasoning about programs written in our language. We will first cover the Hoare-triples defined by Bengtson *et al.* in [3] and then we will cover the new primitives *send*, *recv* and *start*.

The additions we have made to the assertion and specification logic allows the existing logics to be embedded within the new. This means that the axiomatic semantics defined by Bengtson *et al.* still holds. The mechanized proofs are slightly harder to read than the rules we present in this thesis, as we have left all embedding implicit in this thesis. Thus, the rules presented here are equivalent to those from Fig. 4 of [3], while the mechanized proofs contain the explicit `embed`s that we have introduced. The embedding serves to convert assertion predicates of either *asn*, *sasn* or *pasn* into predicates in the *psasn* assertion logic. Because the embedding is trivial – simply drop arguments given to the assertions – the embedability can be automatically inferred.

The axiomatic semantics inherited from the existing language can be seen in Fig. 3.4. We have left out the rules for reasoning about arrays as they are not important to this thesis.

A rule that requires a little introduction is the Frame-rule, which was first proposed by Yang *et al.* in [18]. The frame rule is key to modular reasoning with Separation Logic as it allows more local reasoning. In short, it allows the removal of predicates that talk about different parts of the heap or Session Type map than what the command modifies.

As we have previously mentioned, it is currently not possible to construct a Separation Logic out of our *psasn*-assertion logic that separates both the heap and the Session Type map. Because of this, the mechanized proof for the rule is left admitted.

To reason about programs containing our new primitives, we need to define Hoare-triples covering the *send*, *recv* and *start*-commands. These Hoare-triples are one of the main contributions of this thesis and similarly to our introduction of the operational semantics, we will have a section corresponding to each command.

$$\frac{\{P \wedge e\}\, c\, \{P\}}{\{P\}\, \textbf{while}\, e\, \{c\}\, \{P \wedge \neg e\}}\ \textsc{While} \qquad \frac{\{P\}\, c_1\, \{Q\} \qquad \{Q\}\, c_2\, \{R\}}{\{P\}\, c_1; c_2\, \{R\}}\ \textsc{Seq}$$

$$\frac{\{e \wedge P\}\, c_1\, \{Q\} \qquad \{\neg e \wedge P\}\, c_2\, \{Q\}}{\{P\}\, \textbf{if}\, e\, \{\, c_1\, \}\, \textbf{else}\, \{\, c_2\, \}\, \{Q\}}\ \textsc{Rule-If} \qquad \frac{}{\{P\}\, \text{skip}\, \{P\}}\ \textsc{Skip}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q}{\{P'\}\, c\, \{Q'\} \vdash \{P\}\, c\, \{Q\}}\ \textsc{Consequence} \qquad \frac{P \vdash e}{\{P\}\, \text{assert}\, e\, \{P\}}\ \textsc{Assert}$$

$$\frac{}{\{x.f \hookrightarrow \_\}\, x.f = e\, \{x.f \hookrightarrow e\}}\ \textsc{Write}$$

$$\frac{P \vdash Q * x.f \hookrightarrow \_}{\{P\}\, x.f = e\, \{Q * x.f \hookrightarrow e\}}\ \textsc{Write-Frame} \qquad \frac{}{\{P\{^{e}/_{x}\}\}\, x = e\, \{P\}}\ \textsc{Assign}$$

$$\frac{}{\{P\}\, x = e\, \{\exists v, x = e\{^{v}/_{x}\} \wedge P\{^{v}/_{x}\}\}}\ \textsc{Assign-Fwd}$$

$$\frac{\triangleright \text{C::m}(\bar{p}) \mapsto \{P\}\_\{r.\, Q\} \qquad |\bar{p}| = |\bar{e}|}{\{x = v \wedge P\{^{\bar{p}}/_{\bar{e}}\}\}\, x = \text{C::m}(\bar{e})\, \{Q\{^{r.\bar{p}}/_{x.\bar{e}\{^{v}/_{x}\}}\}\}}\ \textsc{Call-Static}$$

$$\frac{y : C \qquad \triangleright \text{C::m}(\bar{p}) \mapsto \{P\}\_\{r.\, Q\} \qquad |\bar{p}| = |\bar{e}|}{\{x = v \wedge P\{^{\bar{p}}/_{\bar{e}}\}\}\, x = y.m(\bar{e})\, \{Q\{^{r.\bar{p}}/_{x.\bar{e}\{^{v}/_{x}\}}\}\}}\ \textsc{Call-Dynamic}$$

$$\frac{P \vdash \mathbf{y}.f \hookrightarrow e}{\vdash \{P\}\, x = y.f\, \{\exists v,\, x = e\{^{v}/_{x}\} \wedge P\{^{v}/_{x}\}\}}\ \textsc{Read-Fwd}$$

$$\frac{}{\{true\}\, x = \textbf{alloc}\, C\, \{\forall f \in \mathit{fields}(C).\, x.f \hookrightarrow \mathit{null}\}}\ \textsc{Alloc}$$

$$\frac{\forall x \in \mathit{fv}\, R.\, c\ \mathit{does\ not\ modify}\ x}{\{P\}\, c\, \{Q\} \vdash \{P * R\}\, c\, \{Q * R\}}\ \textsc{Frame}$$

Figure 3.4: Hoare-triples inherited from the existing language

3.6.1 Receive

The simplest of the rules is the rule for receiving. Intuitively we want the precondition to be that the channel we are receiving on has a Session Type that starts with a receive. The postcondition should then have the Session Type modified such that it only contains the tail, and we should know that the predicate describing the transferred data holds. However, as visible from the rule given in the theorem below, the postcondition is slightly more complex.

Theorem 3.4 (rule_recv)**.** *The following Hoare-triple for receiving holds.*

$$\frac{x \neq y}{\{\boldsymbol{x} : ?z, \{P\}.T\}\ y = \text{recv}\ x\ \{\exists v, P\{\overline{v/_z}\} \wedge \boldsymbol{y} = v * \boldsymbol{x} : T\{^v/_z\}\}}\ \textsc{Recv}$$

The reason for the more complex postcondition is twofold. Firstly the predicate P needs to have the placeholder substituted for the received value. Thus the predicate on the received data becomes $P\{\overline{\mathbf{y}/_z}\}$ instead of just P. We use a truncating substitution to eliminate any program variables in the predicate that are not the placeholder. Secondly we also want to replace the placeholder variable in the predicates contained within the tail of the Session Type. So we use v as a snapshot of the received value and then substitute the tail of the protocol with this value. This means that the Session Type continuation becomes $T\{^v/_z\}$ instead of simply T. The snapshot is required because the predicates inside the Session Type does not depend on the stack. If the Session Type is substituted with $\mathbf{y}$, and $\mathbf{y}$ later changed value, this change would propagate into the Session Type. By introducing the snapshot, whose predicate does depend on the stack, this problem is avoided.

3.6.2 Send

If the Session Type of x starts with the sending-action of P, then we need to prove that P holds with the placeholder substituted for the value we are sending before we are allowed to perform the send. The postcondition of the triple is identical to that of receive. After sending the value, the object is still present and thus the predicate still holds. As for the Session Type, the same substitution is performed on the tail of the protocol.

Theorem 3.5 (rule_send). *The following Hoare-triple for send is valid.*

$$\frac{}{\begin{array}{c}\{\boldsymbol{x} : !z, \{P\}.T * P\{\overline{y/_z}\} \wedge \mathcal{D}_{sasn}\,(P\{\overline{y/_z}\})\} \\ \text{send}\, x\, y \\ \{\exists v, P\{\overline{y/_z}\} \wedge \boldsymbol{y} = v * \boldsymbol{x} : T\{^v/_z\}\}\end{array}}\ \textsc{Send}$$

Decidability As mentioned under the operational semantics for *send*, we require that the transferred predicate must either hold or not hold, as the frame property of the semantic command can not be proven otherwise. This is the reason why the precondition of our SEND-rule contains the $\mathcal{D}_{\text{sasn}}$-predicate. By requiring a proof of decidability in the Hoare-triple, it is arguable that if you only compile complete, verified programs, the semantics can be implemented by a compiler.

One of the interesting properties of predicates defined in intuitionistic logics is that they are not decidable, unlike in classical logic where the law of the excluded middle holds. So this introduction of decidability might seem unnecessary and limiting. In this section, we will only explain how decidable assertions are defined, and then in Section 6.1.2 we discuss why decidability is a necessary requirement and possible ways to get rid of it.

Definition 3.19 (Decidable). *A predicate is said to be decidable iff for a given state σ, the predicate P either holds or not.*

$$\mathcal{D}_\gamma\, P \triangleq \forall \sigma, P\,\sigma \vee \neg P\,\sigma$$

Where γ denotes the name of the logic which P is written in, e.g. sasn.

Before the SEND-rule can be applied, the decidability of the predicate must be proven. Without rules to reason about decidability of predicates, utilizing the Hoare-triples of our language would be rather hard. So one of the contributions of this thesis is a set of rules that allows the construction of decidability proofs. These rules are given in Figure 3.5

The axioms allow reasoning about many common data structures and their decidability. For instance, it is possible to prove the decidability of the *List*-predicate that we defined in 3.7 using nothing but the rules of Separation Logic and our decidability rules.

Theorem 3.6 (List_decidable). *The List-predicate is decidable for all pointers and logical lists.*

$$\forall p\, xs, \mathcal{D}_{sasn}(List\ p\ xs)$$

$$\frac{}{\mathcal{D}_{\text{sasn}}\,(v.f \hookrightarrow v')}\ \text{D-PointstoI1} \qquad \frac{\mathcal{D}_{\text{sasn}}\,Q}{\mathcal{D}_{\text{sasn}}\,(Q * v.f \hookrightarrow v')}\ \text{D-PointstoI2}$$

$$\frac{\forall a, \mathcal{D}_{\text{sasn}}\,(Q\,a)}{\mathcal{D}_{\text{sasn}}\,(\exists a, Q\,a * v.f \hookrightarrow a)}\ \text{D-ExistsI} \qquad \frac{\mathcal{D}_{\text{sasn}}\,P \qquad \mathcal{D}_{\text{sasn}}\,Q}{\mathcal{D}_{\text{sasn}}\,(P \wedge Q)}\ \text{D-AndI}$$

Figure 3.5: Rules for the construction of decidability proofs

3.6.3 Start

For starting new processes and setting up the communication channel, there are no state-dependent prerequisites. If both the protocol and the desired method to start exist and the method has a startable specification, then the method can be started at any time. After the method has been started, the communication channel has the dual of the protocol used by the started method. The Hoare triple for *start* is thus rather straightforward.

Theorem 3.7 (rule_start)**.** *The following Hoare-triple for starting is valid.*

$$\frac{\triangleright \text{C::m}(a) \mapsto \{a : T\}\_\{r.\ all\_ST : \epsilon\} \qquad p : T}{\{true\}\ x = \text{start C::m}\ p\ \{\boldsymbol{x} : \overline{T}\}}\ \textsc{Start}$$

Because *start* is modeled after method-calls, intuitively you would not expect to have a prerequisite on whether the protocol is available. Instead you might expect that the protocol could be inferred from the specification of the started method. As mentioned earlier, having to statically specify the available protocols is not unheard of, although the axiomatic semantics of our language would be simpler if this could be avoided. The reasoning behind why we need this requirement is discussed in Section 6.1.1.

Chapter 4

Case Studies

To show the applicability of our language, we have performed a set of case studies that highlights the abilities our language possesses.

First we have two variations of a very simple client-server program that serves to illustrate both how our decorated programs look and how the substitution within protocols can serve to propagate information into the remaining protocol.

Finally, we will give a decorated version of our distributed merge sort to verify that the methods live up to the specifications we gave in Chapter 2 and thus that Theorem 2.1 holds.

Some of the decorated programs in this section have been cut short or deliberately kept informal to make them more easily comprehensible. Longer, more formal versions are given in Appendix B. In all of the decorated programs, there are implicit uses of both the Frame- and Consequence-rule whenever any of the Hoare-triples are applied.

4.1 Simple math server

Our first example is a simple math server, which just adds two to a number. It consists of a class, *Sample*, with two methods, *client* and *server*. The method-body of the server can be seen in Fig. 4.1. It asks to receive a number, adds two and then sends it back. The body of *client*, shown in Fig. 4.2, starts a connection with the *server*-method, sends a number and receives the answer.

To formalize the behavior of the server and client method, we have defined a method-specification for both methods. We have chosen the

$a = \text{recv}\ x;$
$b = a + 2;$
$\text{send}\ x\ b$

Figure 4.1: Method body of Sample::server

$x = \text{start Sample::server}\ p;$
$\text{send}\ x\ a;$
$b = \text{recv}\ x$

Figure 4.2: Method body of Sample::client

Session Type $?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon$ to describe the necessary communication between the two processes. To represent the relationship between the input- and output number, we use the globally quantified logical variable n.

$$\begin{aligned} \mathit{Server\_spec} &\triangleq \forall n, \text{Sample::server}(\mathbf{x}) \mapsto \\ &\qquad \{\mathbf{x} : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\}\_\{\mathbf{r}.\, \mathit{all\_ST}\ \epsilon\} \\ \mathit{Client\_spec} &\triangleq \forall n,\ p : ?y, \{\mathbf{y} = n\}.!z, \{\mathbf{z} = n + 2\}.\epsilon \vdash \\ &\qquad \text{Sample::client}(\mathbf{a}) \mapsto \{\mathbf{a} = n\}\_\{\mathbf{b}.\, \mathbf{b} = n + 2\} \end{aligned}$$

With method-bodies and -specifications, we can start proving the correctness of this distributed math service. First, we will show that the method-body of *server* lives up to *Server_spec*:

Decorated Program 4.1: Sample::server living up to *Server_spec*

$\{\mathbf{x} : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\}$
$a\ =\ \text{recv}\ x$
$\{\exists v, (\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{(\mathbf{z} = n + 2)\{^{v}/_{\mathbf{y}}\}\}.\epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}\} \Rightarrow$
$\{\mathbf{a} = n \wedge \mathbf{a} + 2 = \mathbf{a} + 2 * \mathbf{x} : !z, \{\mathbf{z} = n + 2\}.\epsilon\} =$
$\{(\mathbf{a} = n \wedge \mathbf{b} = \mathbf{a} + 2\ * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon)\{^{\mathbf{a}+2}/_{\mathbf{b}}\}\}$
$b\ =\ a + 2$
$\{\mathbf{a} = n \wedge \mathbf{b} = \mathbf{a} + 2\ * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} \Rightarrow$
$\{\mathbf{b} = n + 2\ * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} =$

$\{(\mathbf{z} = n+2)\{^{\mathbf{b}}/_{\mathbf{z}}\} * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon\} \Rightarrow$
$\{(\mathbf{z} = n+2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathcal{D}_{\text{sasn}}\,((\mathbf{z} = n+2)\{^{\mathbf{b}}/_{\mathbf{z}}\}) *$
$\quad \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon\}$
send $x\ b$
$\{\exists v, (\mathbf{z} = n+2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathbf{b} = v * \mathbf{x} :\ \epsilon\} \Rightarrow \{\mathbf{x} :\ \epsilon\} \Rightarrow$
$\{all\_ST\,\epsilon\}$

Two steps in particular need some more explanation: The introduction of $\mathcal{D}_{\text{sasn}}$ and the jump $\mathbf{x} : \epsilon \Rightarrow all\_ST\,\epsilon$. We are allowed to introduce $\mathcal{D}_{\text{sasn}}\,(\mathbf{b} = n+2)$ because it represents decidability between values for which there is decidable equality.
We are allowed to conclude that all protocols have terminated once the protocol for channel $\mathbf{x}$ has terminated because we know that *Sample::server* will be invoked by *start*. And we know that *start* sends along a new Session Type environment with only the single channel $\mathbf{x}$. Since we know that *Sample::client* does not create any new channels, we know that $\mathbf{x}$ is the only channel that exists. Since we know $\mathbf{x}$ has terminated, we know that all the existing channels have terminated.

We also need to show that the *client* method-body implements the *Client_spec*, which the following decorated program shows:

Decorated Program 4.2: Sample::client living up to *Client_spec*

$\{\mathbf{a} = n\}$
$x =$ start Sample::server p
$\{\mathbf{a} = n * \mathbf{x} : \overline{?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon}\} =$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon\} =$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon\} \Rightarrow$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathcal{D}_{\text{sasn}}\,((\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\}) *$
$\quad \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = n+2\}.\epsilon\}$
send $x\ a$
$\{\exists v, (\mathbf{z} = n)\{^{\mathbf{a}}/_{\mathbf{z}}\} \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = n+2\}\{^{v}/_{\mathbf{y}}\}.\epsilon\} =$

$\{\exists v, \mathbf{a} = n \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\}$
$b = \text{recv } x$
$\{\exists v, \mathbf{a} = n \wedge (\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathbf{b} = v * \mathbf{x} : \epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge (\mathbf{b} = n + 2) \wedge \mathbf{b} = v * \mathbf{x} : \epsilon\} \Rightarrow$
$\{\mathbf{a} = n \wedge \mathbf{b} = n + 2 * \mathbf{x} : \epsilon\} \Rightarrow$
$\{\mathbf{b} = n + 2\}$

Again we rely on the decidable equality of values to introduce the term $\mathcal{D}_{\text{sasn}}(\mathbf{a} = n + 2)$ before attempting to send the variable $\mathbf{a}$.

The method specification for *client* never requires that the specification for *server* is defined, and without the specification the method can not be started. As a way to introduce method specifications, we need to introduce the concept of *later*, which is denoted by $\triangleright(.)$. *Later* is known from Gödel-Löb logic and work by Appel *et al.* in [1] and enables both modular reasoning about programs as well as guarded recursion. The intuition is that, under the assumption that iff some specification is proven at a later time, this specification holds [3].

The basic rules for reasoning with later are the Löb- and Weaken-rule shown in Fig. 4.3, and Bengtson *et al.* describe how they are used in the existing language in Section 3.6 of [3]. The basic concept is that the rules are tied to the step-index of the specification logic.

$$\frac{\triangleright P \vdash P}{\vdash P}\ \text{Löb} \qquad \frac{}{P \vdash \triangleright P}\ \text{Weaken}$$

Figure 4.3: The Löb rule for the later modality

Using the Later-rule, we can introduce an assumption that the method-specification for *server* exists, such that the method can be started.

4.1.1 Information Propagation

Instead of relying on the globally quantified variable n to convey the relationship between the received and sent number, we could also use a

program variable in the assertion for the result. So instead of having the sent assertion be $\mathbf{z} = n + 2$, it would be $\mathbf{z} = \mathbf{y} + 2$. The program variable $\mathbf{y}$ refers back to the placeholder of the input, and, as discussed earlier, whenever a send or receive action is performed, the remaining protocol has the current placeholder substituted for the transferred value.

An alternative specification that utilizes this propagation of information could be the following, which is identical to the previous specification except that the Session Type is $?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon$. Note that we still retain the logical variable n in the first predicate. As there are no other restraints for n, it serves only to ensure that the received variable is numerical.

$$
\begin{aligned}
\mathit{Server\_spec}' \triangleq\ & \forall n, \text{Sample::server}(\mathbf{x}) \mapsto \\
& \{\mathbf{x} : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}\_\{\mathit{all\_ST}\ \epsilon\} \\
\mathit{Client\_spec}' \triangleq\ & \forall n,\ p' : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon \vdash \\
& \text{Sample::client}() \mapsto \{\mathbf{a} = n\}\_\{\mathbf{a} = n + 2\}
\end{aligned}
$$

We can reuse the method bodies shown in Figure 4.1 and 4.2, except of course for the protocol given to *start*, which now needs to be p'.
Thus we can prove that the *server* method-body implements the specifications $\mathit{Server\_spec}'$:

Decorated Program 4.3: Sample::server living up to $\mathit{Server\_spec}'$

$\{\mathbf{x} : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}$
$a \ =\ \text{recv}\, x$
$\{\exists v, (\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{(\mathbf{z} = \mathbf{y} + 2)\{^{v}/_{\mathbf{y}}\}\}.\epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = v + 2\}\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}\} \Rightarrow$
$\vdots$
From this point onwards, the decorated program is identical to decorated program 4.1.

For the full decorated program of Sample::server living up to $\mathit{Server\_spec}'$, see decorated program B.1

And we can show that the *client*-body, when it starts the protocol p', lives up to the specification $Client\_spec'$:

Decorated Program 4.4: Sample::client living up to $Client\_spec'$

$\{\mathbf{a} = n\}$
$x = \text{start Sample::server } p'$
$\{\mathbf{a} = n * \mathbf{x} : \overline{?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon}\} =$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\} =$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathcal{D}_{\text{sasn}}\,((\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\}) *$
$\quad \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}$
send $x\ a$
$\{\exists v, (\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}\{^{v}/_{\mathbf{y}}\}.\epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = v + 2\}.\epsilon\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} \Rightarrow$
$\vdots$
From this point onwards, the decorated program is identical to decorated program 4.2.

For the full decorated program of Sample::server living up to $Server\_spec'$, see decorated program B.2

From these decorated programs, it is evident that there is no great difference in the proofs to verify compliance to either a specification where global quantified variables are used or one where information propagation is used.

4.2 Distributed Merge Sort

With the simple example over, we can revisit our main example: Distributed merge sort. The implementation and specification of the example was given in Section 2, but, for convenience, we have replicated the specifications and relevant method-bodies here. As previously men-

tioned, we will not prove the *split* or *merge* methods but instead just rely on their specifications.

$$MSP \triangleq ?\mathbf{i}, \{List\ \mathbf{i}\ xs\}.!\mathbf{o}, \{List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys\}.\epsilon$$

$$\begin{aligned} Split\_spec \triangleq\ & \forall xs, \text{MergeSort::split}(\mathbf{l}) \mapsto \{List\ \mathbf{l}\ xs\}\_ \\ & \{\mathbf{r}.\ \exists r_1\ r_2\ xs_1\ xs_2,\ \mathbf{r}.fst \hookrightarrow r_1 * List\ r_1\ xs_1 \wedge |xs_1| \geq 1 * \\ & \quad \mathbf{r}.snd \hookrightarrow r_2 * List\ r_2\ xs_2 \wedge |xs_2| \geq 1 \wedge xs = xs_1 +\!\!+\ xs_2 \end{aligned}$$

$$\begin{aligned} Merge\_spec \triangleq\ & \forall xs_1\ xs_2\ ys_1\ ys_2, \text{MergeSort::merge}(\mathbf{l1}, \mathbf{l2}) \mapsto \{List\ \mathbf{l1}\ ys_1 * \\ & \quad List\ \mathbf{l2}\ ys_2 \wedge Sorted\_of\ xs_1\ ys_1 \wedge \\ & \quad Sorted\_of\ xs_2\ ys_2\}\_ \\ & \{\mathbf{r}.\ \exists ys,\ List\ \mathbf{r}\ ys \wedge Sorted\_of(xs_1 +\!\!+\ xs_2)\ ys\} \end{aligned}$$

$$MS\_spec \triangleq p : MSP \vdash \text{MergeSort::MS}(x) \mapsto \{x : MSP\}\_\{r.all\_ST : \epsilon\}$$

$$\begin{aligned} Sort\_spec \triangleq\ & p : MSP \vdash \text{MergeSort::sort}(l) \mapsto \{List\ l\ xs\}\_ \\ & \{r.\ \exists ys,\ List\ r\ ys \wedge Sorted\_of\ xs\ ys\} \end{aligned}$$

It is worth noticing that, because *split* needs to return two separate lists, we rely on an anonymous class with the properties *fst* and *snd*. A more thorough specification would also include a specification for a tuple-class and then *Split_spec* could rely on this tuple specification to return two values.

The method-body of the main distributed merge sort method, *MS*, can seen replicated in 4.4.

```
l = recv x;                          xr = start MergeSort::MS p;
if l.length() ≤ 1{                   send xl ll;
  send x l                           send xr lr;
}else{                               sl = recv lr;
  p = MergeSort::split(l);           sr = recv xr;
  ll = p.fst;                        s = MergeSort::merge(sl, sr);
  lr = p.snd;                        send x s;
  xl = start MergeSort::MS p;      }
```

Figure 4.4: Method body of MergeSort::MS

The method *sort* – whose method-body is replicated in Fig. 4.5 – is used to start the whole process, and it does this simply by starting a *MS*-process, sending the lists to *MS* and then receiving the sorted list.

$$x = \text{start}\,\text{MergeSort::MS}\,p; \qquad \text{send}\,x\,l; \qquad r = \text{recv}\,x$$

Figure 4.5: Method body of MergeSort::sort

From the definition of *sort*, you could be led to believe that *MS* could be simplified by replacing the *start*, *send* and *recv* calls inside *MS* with calls to *sort*. Such a simplification would yield a method that behaves identically but lacks a significant property: concurrency. Using calls to *MS* would lead to sequentially sorting the two lists instead of concurrently having the two subprocesses sort the sublists.

As the merge sort example is more extensive than the previous simple math server example, the decorated programs in this section are less explicit. Where the previous examples had multiple small steps between each command, these examples are kept to only two or three. In Appendix B, they are reproduced with the same level of detail as the previous decorated programs.

We start by showing that the method body for *MergeSort::sort* lives up to *Sort_spec*:

Decorated Program 4.5: MergeSort::sort living up to *Sort_spec*

$\{List\,\mathbf{l}\,xs\}$
$x = \text{start}\,\text{MergeSort::MS}\,p$
$\{List\,\mathbf{l}\,xs *$
$\quad \mathbf{x} : \overline{?\mathbf{i}, \{List\,\mathbf{i}\,xs\}.!\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon}\} \Rightarrow$
$\{(List\,\mathbf{i}\,xs)\{^{l}/_{i}\} \wedge \mathcal{D}_{\text{sasn}}\,((List\,\mathbf{l}\,xs)\{^{l}/_{i}\}) *$
$\quad \mathbf{x} : !\mathbf{i}, \{List\,\mathbf{i}\,xs\}.?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\}$
$\text{send}\,x\,l$
$\{(List\,\mathbf{i}\,xs)\{^{l}/_{i}\} * \mathbf{x} : ?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\} \Rightarrow$
$\{\mathbf{x} : ?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\}$
$r = \text{recv}\,x$
$\{(\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\{^{o}/_{r}\} * \mathbf{x} :\ \epsilon\} \Rightarrow$

$\{\exists ys,\ List\ \mathbf{r}\ ys \wedge Sorted\_of\ xs\ ys\}$

For the full decorated program, see decorated program B.3

Again, we use the LATER-rule to introduce the assumption of the method-specification for *MergeSort::MS*, so we are able to start the new process.

For the decorated program that shows that *MS* conforms to the *MS_spec* specification, we introduce shorter names for the session-types. We will use *R* for the protocol that has both receive and send and *S* for the protocol with only the sending-action left. Thus *R* and *S* are defined as the following:

$$R\ xs \triangleq ?\mathbf{i}, \{List\ \mathbf{i}\ xs\}.!\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys\}.\epsilon$$
$$S\ xs \triangleq !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys\}.\epsilon$$

With these protocol-names, we can confirm that *MS* does conform to the *MS_spec* specification with the following decorated program:

Decorated Program 4.6: MergeSort::Sort living up to *Sort_spec*

$\{\mathbf{x} :\ R\ xs\}$
$l = \text{recv}x$
$\{(List\ \mathbf{i}\ xs)\{^{i}/_{l}\} * \mathbf{x} : S\ rs\} \Rightarrow$
$\{List\ \mathbf{l}\ xs * \mathbf{x} : S\ rs\}$
if $\mathbf{l}.length() \leq 1\{$
  $\{List\ \mathbf{l}\ xs \wedge \mathbf{l}.length() \leq 1 * \mathbf{x} : S\ rs\} \Rightarrow$
  $\{List\ \mathbf{l}\ xs * \mathbf{x} :\ S\ rs \wedge Sorted\_of\ xs\ xs\} \Rightarrow$
  $\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{o}/_{l}\} \wedge$
    $\mathcal{D}_{\text{sasn}}\left((\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{o}/_{l}\}\right) * \mathbf{x} :\ S\ rs\}$
  send $x\ l$
  $\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{o}/_{l}\} * \mathbf{x} :\ \epsilon\} \Rightarrow$
  $\{\mathbf{x} :\ \epsilon\} \Rightarrow \{all\_ST\ \epsilon\}$
$\}$ else $\{$

$\{List\,\mathbf{l}\,xs * \mathbf{x} : S\,rs \wedge \mathbf{l}.length() > 1\} \Rightarrow$
$\{List\,\mathbf{l}\,xs * \mathbf{x} : S\,rs\}$
$p = \text{MergeSort::split}\,l$
$\{\mathbf{p}.fst \mapsto r_1 * List\,r_1\,xs_1 * \mathbf{p}.snd \mapsto r_2 * List\,r_2\,xs_2$
$\quad \wedge\ xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs\}$
$ll = p.fst$
$\{List\,\mathbf{ll}\,xs_1 * \mathbf{p}.snd \mapsto r_2 * List\,r_2\,xs_2$
$\quad xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs\}$
$lr = p.snd$
$\{List\,\mathbf{ll}\,xs_1 * List\,\mathbf{lr}\,xs_2 \wedge xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs\}$
$xl = \text{start MergeSort::MS}\,p$
$\{List\,\mathbf{ll}\,xs_1 * List\,\mathbf{lr}\,xs_2 \wedge xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs * \mathbf{xl} : \overline{R\,xs_1}\}$
$xr = \text{start MergeSort::MS}\,p$
$\{List\,\mathbf{ll}\,xs_1 * List\,\mathbf{lr}\,xs_2 \wedge xs_1 +\!\!+\ xs_2 = xs *$
$\quad \mathbf{x} : S\,xs * \mathbf{xl} : \overline{R\,xs_1} * \mathbf{xr} : \overline{R\,xs_2}\} \Rightarrow$
$\{(List\,\mathbf{i}\,xs_1)\{^i/_{ll}\} \wedge \mathcal{D}_{\text{sasn}}\,((List\,\mathbf{i}\,xs_1)\{^i/_{ll}\}) * List\,\mathbf{lr}\,xs_2 \wedge$
$\quad xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs * \mathbf{xl} : \overline{R\,xs_1} * \mathbf{xr} : \overline{R\,xs_2}\} \Rightarrow$
$\text{send}\,xl\,ll$
$\{(List\,\mathbf{i}\,xs_1)\{^i/_{ll}\} * List\,\mathbf{lr}\,xs_2 \wedge xs_1 +\!\!+\ xs_2 = xs *$
$\quad \mathbf{x} : S\,xs * \mathbf{xl} : \overline{S\,xs_1} * \mathbf{xr} : \overline{R\,xs_2}\} \Rightarrow$
$\{(List\,\mathbf{i}\,xs_2)\{^i/_{lr}\} \wedge \mathcal{D}_{\text{sasn}}\,((List\,\mathbf{i}\,xs_2)\{^i/_{lr}\}) \wedge$
$\quad xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs * \mathbf{xl} : \overline{R\,xs_1} * \mathbf{xr} : \overline{R\,xs_2}\} \Rightarrow$
$\text{send}\,xr\,lr$
$\{(List\,\mathbf{i}\,xs_2)\{^i/_{lr}\} \wedge xs_1 +\!\!+\ xs_2 = xs *$
$\quad \mathbf{x} : S\,xs * \mathbf{xl} : \overline{S\,xs_1} * \mathbf{xr} = \overline{S\,xs_2}\} \Rightarrow$
$\{xs_1 +\!\!+\ xs_2 = xs * \mathbf{x} : S\,xs * \mathbf{xl} : \overline{S\,xs_1} * \mathbf{xr} = \overline{S\,xs_2}\} \Rightarrow$
$sl = \text{recv}\,xl$
$\{(\exists ys_1,\, List\,\mathbf{o}\,ys_1 \wedge Sorted\_of\,xs_1\,ys_1)\{^o/_{sr}\} \wedge xs_1 +\!\!+\ xs_2 = xs *$
$\quad \mathbf{x} : S\,xs * \mathbf{xl} = \epsilon * \mathbf{xr} = \overline{S\,xs_2}\} \Rightarrow$
$\{List\,\mathbf{sr}\,ys_1 \wedge Sorted\_of\,xs_1\,ys_1 \wedge xs_1 +\!\!+\ xs_2 = xs *$
$\quad \mathbf{x} : S\,xs * \mathbf{xl} = \epsilon * \mathbf{xr} = \overline{S\,xs_2}\}$

$sr = \text{recv}\ xr$
$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1 * (\exists ys_2,\ List\ \mathbf{o}\ ys_2 \wedge$
$Sorted\_of\ xs_2\ ys_2)\{^o/_{sr}\} \wedge xs_1 +\!\!+ xs_2 = xs *$
$\mathbf{x} : S\ xs\ * \mathbf{xl} : \epsilon * \mathbf{xr} : \epsilon\} \Rightarrow$
$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1 * List\ \mathbf{o}\ ys_2\ \wedge Sorted\_of\ xs_2\ ys_2 \wedge$
$xs_1 +\!\!+ xs_2 = xs * \mathbf{x} : S\ xs\ * \mathbf{xl} : \epsilon * \mathbf{xr} : \epsilon\}$
$s = \text{MergeSort::merge}\ sl\ sr$
$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1 * List\ \mathbf{o}\ ys_2\ \wedge Sorted\_of\ xs_2\ ys_2 \wedge$
$xs_1 +\!\!+ xs_2 = xs * \mathbf{x} : S\ xs\ * \mathbf{xl} : \epsilon * \mathbf{xr} : \epsilon *$
$List\ \mathbf{s}\ ys \wedge Sorted\_of\ (xs_1 +\!\!+ xs_2)\} \Rightarrow$
$\{List\ \mathbf{s}\ ys \wedge Sorted\_of\ xs\ ys * \mathbf{x} : S\ xs * \mathbf{xl} : \epsilon\ * \mathbf{xr} : \epsilon\} \Rightarrow$
$\{(\exists ys,\ List\ \mathbf{o}\ xs \wedge Sorted\_of\ xs\ ys)\{^o/_s\} \wedge$
$\mathcal{D}_{\exists ys, \text{sasn}}\ ((List\ \mathbf{o}\ xs \wedge Sorted\_of\ xs\ ys)\{^o/_s\}) *$
$\mathbf{x} : S\ xs * \mathbf{xl} : \epsilon\ * \mathbf{xr} : \epsilon\}$
$\text{send}\ x\ s$
$\{(\exists ys,\ List\ \mathbf{o}\ xs \wedge Sorted\_of\ xs\ ys)\{^o/_s\} *$
$\mathbf{x} : \epsilon * \mathbf{xl} : \epsilon\ * \mathbf{xr} : \epsilon\}$
$\{\mathbf{x} = \epsilon * \mathbf{xl} = \epsilon * \mathbf{xr} :\ \epsilon\} \Rightarrow \{all\_ST\ \epsilon\}$
$\}$
$\{all\_ST\ \epsilon\}$

We once again use Löb-rule to introduce the assumption that the called methods have the method-specifications we desire. In order to prove that *MS_spec* holds we need to know that *MS_spec* holds, in order to recurse. Because the rule is tied to the step-index, using later basically turns the proof into an induction proof over the step-indexes.

In the appendix, decorated program B.4 has all reduction steps for verifying the simpler, serial implementation that uses calls to *sort*. The reason for only having a sequential version as a more formal decorated program is readability. With the actions between the two subprocesses being interleaved, it makes it hard to use the frame-rule to shorten the predicates inside the Hoare-triples. Without the interleaving, everything but one list can be framed out while that list is sorted. So while proving

that each list can be recursively sorted, the proof-context only contains predicates relevant to that list. With the interleaving, each command would require a new framing to remove the predicates unnecessary for that command.

Chapter 5

Related Work

We are aware of no other work that integrates Session Types in Hoare-logic and Separation Logic in the way we are proposing. But there has been related work done that integrates Session Types and Separation Logic. This chapter aims to summarize some of this work.

Hussain, O'Hearn and Petersen have done some work on integrating a subset of Session Types into a very small language based on Concurrent Separation Logic (CSL) [11]. They first define a simple language containing only sending, receiving and parallel composition which they type using the same subset of Session Types that we do – i.e., only sending and receiving messages.

They then define an imperative language based on CSL, where the pre- and postconditions in their Hoare-triples are Session Type typing judgements. They then give a translation method that can translate any valid program in their Session Type subset into an equivalent imperative program.

Their result shows that it is possible to use the shared-memory approach of CSL to embed the message-passing based Session Types logic.

The main difference between the work of Hussain *et al.* and this thesis is that their approach will never work across computers. They rely on shared memory to deliver their messages, which limits the applicability to communication within a single computer. This thesis proposes no such constraints, and it would be possible to implement our model in a way such that it transfers data between computers.

Chen and Honda present a different take on the same goal of proving correctness of distributed programs in [4]. Instead of trying to integrate reasoning about protocols into a large framework for reasoning about state, – using e.g. Separation Logic – Chen and Honda integrate reasoning about state into their framework for reasoning about protocols: choreographies and Session Types.

They propose introducing system-level observers – i.e. runtime monitors – that verify that specifications embedded within the protocols hold. These observers maintain a state which allows for reasoning across actions within a protocol as well as information exchanged in different protocols within the same choreography.

The difference between our proposed model and that of Chen *et al.* is quite stark. Our model allows proving functional correctness of distributed programs, while the runtime monitors introduced by Chen *et al.* are simply able to observe whether or not the communication is valid as it happens. Furthermore, the predicates used to describe the states of communication is less expressive than the assertion logic we have defined.

Chapter 6

Discussion

6.1 Discussion

6.1.1 The necessity of the protocol map

The language we have proposed in this thesis currently needs to know all the protocols of all methods which we later *start*. As previously mentioned, having to explicitly state each protocol is not uncommon amongst Session Types implementations. Session-J, for instance, has the same requirement. However, why was it necessary to introduce this requirement? If we compare it with method-calls – which is what our model of *start* is based upon – there is no such requirement. The axiomatic semantics of method-calls is able to call methods without knowing their specifications, but *start* needs the Session Type before we can specify its operational semantic.

The difference is that method-calls should function as if the method-body was inlined instead of the method call – sans renaming of parameters. So the operational semantics of method calls transfer the (partial) stack to the method body and then continues with the stack extended with the return value. With *start*, the understanding is that the started method should run in parallel. It is thus not important what changes to the stack or heap that the started method does, because it does not need to be passed back to the process that issued the *start*-call. Once *start* has been called, information can only be sent back and forth between the two processes by *send* or *recv* and only as long as it follows the protocol of the instantiated channel. The operational semantics of *start* then extends its Session Type map with the dual of the method's protocol,

which instantiates the channel to the started process. Thus before the channel can be instantiated for either the started or the starting process, the protocol must be known.

Session types describe the remaining actions of each channel's protocol, where Hoare logic places requirements on the precondition and describes the resulting state. That means that there is a fundamental difference in the direction of the reasoning. In the proof of Rule-Start, any use of the method specification for the startable method requires proving the precondition $(a : T)$ and will only yield the fairly useless proof of $all\_ST : \epsilon$. Similarly, proving the postcondition of the rule requires proving $x : \overline{T}$. Thus both cases requires proving a typing judgement where the only context available is the precondition *true*, which is of no help. If we contrast this with method calls, the precondition P of the triple proves the precondition of the method specification, and the postcondition of the method specification proves the postcondition of the triple. Because we need proof of the Session Type typing judgement to both use the method specification and to prove the triples postcondition, we have been unable to automatically infer the required Session Type.

6.1.2 Limited to decidable propositions

The biggest limitation in our language is that it requires decidable assertions within protocols. To apply the *send*-rule, the assertion covering the data we want to send must be decidable. As explained in Section 3.4.1, this is to satisfy the frame property. To prove the frame property, it must be known if removing the frame from the combined heap will cause a memory fault. If a memory fault occurs, then the command would not be safe, but, because we have an assumption of safety, this leads to a contradiction, thus proving that no memory fault could have occurred.

In the simpler commands, such as *read*, it is very explicit when the command requires memory from the added frame. For *read*, this is a case-analysis on heap-membership. If the requested object was outside the frame, the frame property holds. However, if the requested object was in the frame, then we have a memory fault and the above mentioned contradiction on safety.

For sending, we have no inherent property to do a case analysis on. To prove that the command has the frame property, we need to prove that the assertion holds without the heap being extended by a frame. However, we have no way of knowing if the assertion covers the frame

or not. Our solution was to require assertions to be decidable, which enables a case-analysis on whether or not the assertion holds in the frame-less heap.

Another possible solution would be to introduce the notion of *marshaling*. A marshaling consists of a pointer to an object, as well as the subheap that includes the referenced data structure. Any values found by recursive walk of any referenced data structures are also included in the marshaling. In the end, the marshaling will contain the entire memory footprint of an object and all its children. When a pointer is to be sent on a channel, the subheap that needs to be transferred is the marshaling of the object pointed to by the transferred variable. A memory fault would be triggered if the marshaling overlapped with the frame.

The assertion would need to hold with a singleton stack consisting only of the variable being sent and the heap resulting from the marshaling. However, having only the starting point of the marshaling in the stack does not limit the assertion from being able to mention things outside of the marshaled data structure. A *pointsto*-predicate could use a heap-address instead of a pointer from the stack, which enables mentioning addresses outside of the marshaled data structure. To counter this, the notion of marshalability could be introduced. A predicate would be marshalable with respects to a variable if the predicate holds in the marshaled heap starting from the variable.

The reason why marshaling was not chosen over decidability was that we were unsuccessful in defining a marshaling that worked with our assertion logic. Untyped marshaling, which does not utilize knowledge of the object's structure, is not upwards closed on heaps. Because the untyped marshaling simply copies over the structure present in the heap, the result can change if the extended heap contains additional data for the data structure. With statically typed marshaling, which requires all of the data structure to be present, we were unable to construct rules for building marshalings that were compatible with Separation Logic. As soon as a *pointsto*-predicate was added to a marshaling, the memory footprint of the marshaling would become the entire object. So if another *pointsto*-predicate for a different field is available, there is a contradiction because the heaps between the predicate and the marshaling can not be disjoint.

6.2 Further Work

There are many interesting directions this project could take. This section attempts to describe some of the most obvious.

Marshaling Firstly, the marshaling described in Section 6.1.2 should be investigated more. During this thesis, marshaling was attempted and, although unsuccessful then, we believe a version of marshaling can be defined in such a way that it could remove the limitations of decidability. We attempted both an untyped and a statically typed marshaling but have later discovered a dynamically typed definition to marshaling that might be successful. In short, the marshaling should keep track of the fields that have been added to each marshaled object. This way the marshaling result stays the same if the heap is extended, and the memory footprint of the marshaling can be kept to a minimum.

Branching Ordinary Session Types have support for branching within protocols, whereas our current implementation does not. By extending our integration to include the *branch* and *choice*, we could verify communication corresponding to much more complicated protocols. The language primitive for branch would take a list of labels and commands, and the Hoare-triple for each command would have a precondition consisting of the protocol corresponding to the label. For choice, the postcondition would simply be the protocol corresponding to the label.

Intuitively the Hoare-triples should be similar to

$$\frac{\exists i, l = l_i}{\{x : \oplus\langle l_1 : T_1, \ldots, l_n : T_n\rangle\}\ \text{select}\ x\ l\ \{x : T_i\}}\ \textsc{Rule-Choice}$$

$$\frac{\{x : T_1\}\ c_1\ \{Q\} \ldots \{x : T_n\}\ c_n\ \{Q\}}{\{x : \&\langle l_1 : T_1, \ldots, l_n : T_n\rangle\}\ \text{offer}\ \{l_1 : c_1, \ldots, l_n : c_n\}\ \{Q\}}\ \textsc{Rule-Branch}$$

Delegation Adding support for delegation would also allow for the verification of a larger set of programs. With delegation, a program does not have to finish a protocol if it has a connection to another party that is capable of finishing it.

The standard definition of Session Types does this by extending the types of data send and receive actions can send to also include channels and their protocol. Our implementation would need new primitives

for two reasons: Firstly, the assertion logic we use in the predicates inside Session Types does not allow for reasoning about the current active channels. But equally important, the Hoare-triple for sending retains the sent predicate, which is against the intentions of delegation. Once a channel is delegated, it is no longer an active channel of the sender. The operational semantics of sending a channel would thus have to remove knowledge of the delegated channel from the session type map. The receiving rule would have a Hoare-triple similar to Rule-Recv, but sending a delegate would require a triple like

$$\frac{}{\{x : ![S].T \wedge y : S\}\ \text{send\_c}\ x\ y\ \{x : T \wedge y = \mathit{null}\}}\ \textsc{Rule-Send-C}$$

where $![S].T$ is the protocol to send a delegate.

Fixing the Frame Rule As we mentioned in Section 3.3, we have been unable to mechanize the proof the Frame-rule, as the underlying framework our language depends on does not support building a Separation Logic over both the heap as well as the Session Type map. To construct such a logic Charge! – the underlying framework – can be extended with type-classes that allow the construction of a Separation Logic from a Separation Algebra and an existing Separation Logic. Once such a Separation Logic can be built, proving the rule is trivial, as the proof relies solely on the frame-property which we have extended to frame both the heap and Session Type map already.

Chapter 7

Conclusion

This thesis presents an integration between Session Types and Hoare-logic that allows for the verification of distributed programs. Using the example of distributed merge sort, the functional correctness is shown using step-by-step verification. This shows that our integration enables verification of the functional correctness of a recursive, distributed program implemented in our language.

We have taken an existing implementation of a language and extended it with sending, receiving and a notion of Session Types. The correctness of our extended language is proven using mechanized proofs implemented in Coq. By building on top of an existing language, we believe we have built a good foundation for verifying distributed programs. The inherited Hoare-triples from the existing language enables the verification of any sequential sections of programs. The distributed aspects of programs can be encoded using our *send*, *recv* and *start* primitives. The communication between processes can then be specified using Session Types and later verified using our added Hoare-triples.

Bibliography

[1] Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007.

[2] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! In *Interactive Theorem Proving*, pages 315–331. Springer, 2012.

[3] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In *Interactive Theorem Proving*, pages 22–38. Springer, 2011.

[4] Tzu-Chun Chen and Kohei Honda. Specifying stateful asynchronous properties for distributed programs. In *CONCUR 2012–Concurrency Theory*, pages 209–224. Springer, 2012.

[5] Robert Dockins, Aquinas Hobor, and Andrew W Appel. A fresh look at separation algebras and share accounting. In *Programming Languages and Systems*, pages 161–177. Springer, 2009.

[6] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[7] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[8] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer, 1998.

[10] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008–Object-Oriented Programming*, pages 516–541. Springer, 2008.

[11] Akbar Hussain, Peter W O´Hearn, and Rasmus L Petersen. On separation, session types and algebra.

[12] DE Knuth. Vol. 3: Sorting and searching. *Addison-Wesley series in computer science*, 1973.

[13] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[14] Jacques-Louis Lions et al. Ariane 5 flight 501 failure, 1996.

[15] Hannes Mehnert and Jesper Bengtson. Kopitiam–a unified ide for developing formally verified java programs. Technical report, Technical Report ITU-TR-2013-167, IT University of Copenhagen, 2013.

[16] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[17] John C Reynolds. An introduction to separation logic. *Engineering Methods and Tools for Software Safety and Security*, pages 285–310, 2008.

[18] Hongseok Yang and Peter O´Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, pages 402–416. Springer, 2002.

Appendix A

Source Code

A.1 Access to Source Code

The source code for the implementation of our language, as well as our mechanized proofs are available on GitHub. It can be found in the branch `fangel` of the repository `https://github.com/jesper-bengtson/Java`.
A direct link to the relevant branch is
`https://github.com/jesper-bengtson/Java/tree/fangel`.

At the time of this thesis, the latest commit was
`a079d8b290d54fdd8c235d4002f7164383a774ed`.

Charge! The language is built using the Charge!-framework which can be found at the repository
`https://github.com/jesper-bengtson/Charge`
Our language has been built against the framework at commit
`2eaed7666da6074d2e2354ef85ad216b39cf15d8`

A.2 Admitted Proofs

The existing language by Bengtson *et al.* contained a list of admitted proofs, which we will not enumerate here. The only admitted yet applied lemma that we introduced is `heap_eq_dec`

$$heap\_eq\_dec \triangleq \forall (a\ b : heap),\ \{a === b\} + \{a =/= b\}$$

which is the decidability lemma for heap equivalence. It serves to allow case-analysis on whether or not two heaps are equivalent. Two heaps are defined as being equivalent if the object- and array-sections of the heap are equivalent. And equivalence on maps is defined as the maps having equal domains and every key within the domain having equal values.

Intuitively, it is decidable whether or not heaps are equivalent, but, because of the implementation of maps, we have been unable to produce a proof of this.

There are some Hoare-triples, which are not used in the thesis that has been left unproven. The rule for allocating object was left unproven by Bengtson *et al.*, and then additionally we have left Arr-Write unproven.

The most important rule that we have left unproven is the Frame-rule. As mentioned earlier, this is because of a shortcoming in the underlying framework, Charge!, as it is not currently possible to define a Separation Logic that is separate on both the heap and Session Type map. See Section 6.2 for more information.

Appendix B

Decorated Programs

Decorated Program B.1: Sample::server living up to *Server_spec'*

$\{\mathbf{x} : ?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}$
$a \;=\; \text{recv}\; x$
$\{\exists v, (\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{(\mathbf{z} = \mathbf{y} + 2)\{^{v}/_{\mathbf{y}}\}\}.\epsilon\} =$
$\{\exists v, \mathbf{a} = n * \mathbf{a} = v * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = v + 2\}\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}\} \Rightarrow$
$\{\mathbf{a} = n \wedge \mathbf{a} + 2 = \mathbf{a} + 2 * \mathbf{x} : !z, \{\mathbf{z} = n + 2\}.\epsilon\} =$
$\{(\mathbf{a} = n \wedge \mathbf{b} = \mathbf{a} + 2 * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon)\{^{\mathbf{a}+2}/_{\mathbf{b}}\}\}$
$b \;=\; a + 2$
$\{\mathbf{a} = n \wedge \mathbf{b} = \mathbf{a} + 2 * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} \Rightarrow$
$\{\mathbf{b} = n + 2 * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} =$
$\{(\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\} * \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\}$
$\{(\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathcal{D}_{\text{sasn}}\,((\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\}) *$
$\quad \mathbf{x} : !\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\}$
$\text{send}\; x\; b$
$\{\exists v, (\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathbf{b} = v * \mathbf{x} : \epsilon\} \Rightarrow \{\mathbf{x} : \epsilon\}$
$\{all\_ST\,\epsilon\}$

Decorated Program B.2: Sample::client living up to $Client\_spec'$

$\{\mathbf{a} = n\}$
$x = \text{start}\,\text{Sample::server}'\ p'$
$\{\mathbf{a} = n * \mathbf{x} : \overline{?\mathbf{y}, \{\mathbf{y} = n\}.!\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon}\} =$
$\{\mathbf{a} = n * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\} =$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} * \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\} \Rightarrow$
$\{(\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\} \wedge \mathcal{D}_{\text{sasn}}\,((\mathbf{y} = n)\{^{\mathbf{a}}/_{\mathbf{y}}\}) *$
$\quad \mathbf{x} : !\mathbf{y}, \{\mathbf{y} = n\}.?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}.\epsilon\}$
$\text{send}\,x\,a$
$\{\exists v, (\mathbf{z} = n)\{^{\mathbf{a}}/_{\mathbf{z}}\} \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = \mathbf{y} + 2\}\{^{v}/_{\mathbf{y}}\}.\epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge \mathbf{a} = v * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = v + 2\}.\epsilon\} \Rightarrow$
$\{\mathbf{a} = n * \mathbf{x} : ?\mathbf{z}, \{\mathbf{z} = n + 2\}.\epsilon\} \Rightarrow$
$b = \text{recv}\,x$
$\{\exists v, \mathbf{a} = n \wedge (\mathbf{z} = n + 2)\{^{\mathbf{b}}/_{\mathbf{z}}\} \wedge \mathbf{b} = v * \mathbf{x} : \epsilon\} =$
$\{\exists v, \mathbf{a} = n \wedge (\mathbf{b} = n + 2) \wedge \mathbf{b} = v * \mathbf{x} : \epsilon\} \Rightarrow$
$\{\mathbf{a} = n \wedge \mathbf{b} = n + 2 * \mathbf{x} : \epsilon\} \Rightarrow$
$\{\mathbf{b} = n + 2\}$

Decorated Program B.3: MergeSort::Sort living up to $Sort\_spec$

$\{List\,\mathbf{l}\,xs\}$
$x = start\,\text{MergeSort::MS}\,p$
$\{List\,\mathbf{l}\,xs\ *$
$\quad \mathbf{x} : \overline{?\mathbf{i}, \{List\,\mathbf{i}\,xs\}.!\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon}\} =$
$\{List\,\mathbf{l}\,xs\ *$
$\quad \mathbf{x} : !\mathbf{i}, \{List\,\mathbf{i}\,xs\}.?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\} \Rightarrow$
$\{(List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\}\ *$
$\quad \mathbf{x} : !\mathbf{i}, \{List\,\mathbf{i}\,xs\}.?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\} \Rightarrow$

$\{(List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\} \wedge \mathcal{D}_{\text{sasn}}\,((List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\}) *$
$\quad \mathbf{x} : !\mathbf{i}, \{List\,\mathbf{i}\,xs\}.?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\}$
$send\ x\ l$
$\{\exists v, (List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\} \wedge \mathbf{l} = v *$
$\quad \mathbf{x} : ?\mathbf{o}, \{(\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\{^{v}/_{\mathbf{i}}\}\}.\epsilon\} \Rightarrow$
$\{\exists v, (List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\} \wedge \mathbf{l} = v *$
$\quad \mathbf{x} : ?\mathbf{o}, \{(\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\{^{v}/_{\mathbf{i}}\}\}.\epsilon\} \Rightarrow$
$\{List\,\mathbf{l}\,xs * \mathbf{x} : ?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\} \Rightarrow$
$\{\mathbf{x} : ?\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\}$
$r = recv\ x$
$\{\exists v, (\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\{^{\mathbf{r}}/_{\mathbf{o}}\} \wedge \mathbf{r} = v * \mathbf{x} : \epsilon\} \Rightarrow$
$\{(\exists ys,\ List\,\mathbf{r}\,ys \wedge Sorted\_of\,xs\,ys) * \mathbf{x} : \epsilon\} \Rightarrow$
$\{\exists ys,\ List\,\mathbf{r}\,ys \wedge Sorted\_of\,xs\,ys\}$

Decorated Program B.4: MergeSort::MS living up to *MS_spec*

$\{\mathbf{x} : ?\mathbf{i}, \{List\,\mathbf{i}\,xs\}.!\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys\}.\epsilon\}$
$l = \text{recv}\ x$
$\{\exists v, (List\,\mathbf{i}\,xs)\{^{\mathbf{l}}/_{\mathbf{i}}\} \wedge \mathbf{l} = v *$
$\quad \mathbf{x} : !\mathbf{o}, \{(\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\{^{v}/_{\mathbf{i}}\}\}.\epsilon\} \Rightarrow$
$\{List\,\mathbf{l}\,xs * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\}.\epsilon\}$
$\text{if}\ \mathbf{l}.\text{length}() \leq 1\{$
$\quad \{List\,\mathbf{l}\,xs * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\}.\epsilon \wedge$
$\quad\ \mathbf{l}.\text{length}() \leq 1\} \Rightarrow$
$\quad \{List\,\mathbf{l}\,xs * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\}.\epsilon \wedge$
$\quad\ Sorted\_of\,xs\,xs\} \Rightarrow$
$\quad \{\exists ys,\ List\,\mathbf{l}\,ys * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\}.\epsilon$
$\quad\ \wedge\ Sorted\_of\,xs\,ys\} \Rightarrow$
$\quad \{\exists ys,\ (List\,\mathbf{l}\,ys \wedge Sorted\_of\,xs\,ys) *$
$\quad\ \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\,\mathbf{o}\,ys \wedge Sorted\_of\,xs\,ys)\}.\epsilon\} =$

$\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{l}}/_{\mathbf{o}}\} *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
$\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{l}}/_{\mathbf{o}}\} \wedge$
$\mathcal{D}_{\text{sasn}}\ ((\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{l}}/_{\mathbf{o}}\})\ *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
send $x\ l$
$\{\exists v,\ (\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{l}}/_{\mathbf{o}}\}\ \wedge \mathbf{l} = v * \mathbf{x} :\ \epsilon\} \Rightarrow$
$\{\mathbf{x} :\ \epsilon\} \Rightarrow$
$\{all\_ST\ \epsilon\}$
}else{
$\{List\ \mathbf{l}\ xs * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon \wedge$
$\wedge\, \mathbf{l}.\text{length}() \not\leq 1\} \Rightarrow$
$\{List\ \mathbf{l}\ xs * \mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
$p = \text{MergeSort::split}\ l$
$\{(\exists r_1\ r_2\ xs_1\ xs_2,\ \mathbf{r}.fst \mapsto r_1 * List\ r_1\ xs_1 * \mathbf{r}.snd \mapsto r_2 *$
$List r_2 xs_2 \wedge xs = xs_1 +\!\!+\ xs_2)\{^{\mathbf{p}}/_{\mathbf{r}}\} *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} \Rightarrow$
$\{\mathbf{p}.fst \mapsto r_1 * List\ r_1\ xs_1 * \mathbf{p}.snd \mapsto r_2 * List\ r_2\ xs_2 \wedge$
$xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
$\Rightarrow$ (framing)
$\{\mathbf{p}.fst = r_1 * List\ r_1\ xs_1\}$
$l = p.fst$
$\{\exists v, \mathbf{l} = r_1\{^{v}/_{\mathbf{l}}\} \wedge (\mathbf{p}.fst = r_1 * List\ r_1\ xs_1)\{^{v}/_{\mathbf{l}}\}\} \Rightarrow$
$\{\mathbf{l} = r_1 \wedge \mathbf{p}.fst = r_1 * List\ r_1\ xs_1\} \Rightarrow$
$\{List\ \mathbf{l}\ xs_1\} \Rightarrow$
$\{(List\ l\ xs_1)\{^{\mathbf{l}}/_{l}\}\}$
$sl = \text{MergeSort::sort}(l)$
$\{(\exists ys,\ List\ \mathbf{r}\ ys \wedge Sorted\_of\ xs_1\ ys)\{^{\mathbf{sl}}/_{\mathbf{r}}\}\} =$
$\{\exists ys,\ List\ \mathbf{sl}\ ys \wedge Sorted\_of\ xs_1\ ys\} \Rightarrow$
$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1\} \Rightarrow$

$\Rightarrow$ (end framing)

$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1 * \mathbf{p}.snd \mapsto r_2 * List\ r_2\ xs_2 \wedge$
$xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$

$\Rightarrow$ (framing)

$\{\mathbf{p}.snd = r_2 * List\ r_2\ xs_2\}$
$r = p.snd$
$\{\exists v,\ \mathbf{r} = r_2\{^{v}/_{\mathbf{r}}\} \wedge (\mathbf{p}.snd = r_2 * List\ r_2\ xs_1)\{^{v}/_{\mathbf{r}}\}\} \Rightarrow$
$\{\mathbf{r} = r_2 \wedge \mathbf{p}.snd = r_2 * List\ r_2\ xs_2\} \Rightarrow$
$\{List\ \mathbf{r}\ xs_2\} \Rightarrow$
$\{(List\ l\ xs_2)\{^{\mathbf{r}}/_{l}\}\}$
$sr = \text{MergeSort::sort}(r)$
$\{(\exists ys,\ List\ \mathbf{r}\ ys \wedge Sorted\_of\ xs_2\ ys)\{^{\mathbf{sr}}/_{\mathbf{r}}\}\} =$
$\{\exists ys,\ List\ \mathbf{sr}\ ys \wedge Sorted\_of\ xs_2\ ys_2\} \Rightarrow$
$\{List\ \mathbf{sr}\ ys_2 \wedge Sorted\_of\ xs_2\ ys_2\} \Rightarrow$

$\Rightarrow$ (end framing)

$\{List\ \mathbf{sl}\ ys_1 \wedge Sorted\_of\ xs_1\ ys_1 * List\ \mathbf{sr}\ ys_2 \wedge$
$Sorted\_of\ xs_2\ ys_2 \wedge xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} \Rightarrow$
$\{List\ \mathbf{sl}\ ys_1 * List\ \mathbf{sr}\ ys_2 \wedge Sorted\_of\ xs_1\ ys_1 \wedge$
$Sorted\_of\ xs_2\ ys_2 \wedge xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} =$
$\{(List\ l1\ ys_1 * List\ l2\ ys_2 \wedge Sorted\_of\ xs_1\ ys_1 \wedge$
$Sorted\_of\ xs_2\ ys_2)\{^{[\mathbf{sl},\mathbf{sr}]}/_{[l1,l2]}\} \wedge xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
$s = \text{MergeSort::merge}(\mathbf{sl}, \mathbf{sr})$
$\{(\exists ys,\ List\ \mathbf{r}\ ys \wedge Sorted\_of(xs_1 +\!\!+\ xs_2)\ ys)\{^{\mathbf{s}}/_{\mathbf{r}}\} \wedge$
$xs = xs_1 +\!\!+\ xs_2 *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} \Rightarrow$
$\{\exists ys,\ List\ \mathbf{s}\ ys \wedge Sorted\_of xs\ ys *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} =$

$\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{s}}/_{\mathbf{o}}\} *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\} \Rightarrow$
$\{(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{s}}/_{\mathbf{o}}\} \wedge$
$\mathcal{D}_{\text{sasn}}\,(\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{s}}/_{\mathbf{o}}\}) *$
$\mathbf{x} : !\mathbf{o}, \{\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\}.\epsilon\}$
send $x\ s$
$\{\exists v, (\exists ys,\ List\ \mathbf{o}\ ys \wedge Sorted\_of\ xs\ ys)\{^{\mathbf{s}}/_{\mathbf{o}}\} \wedge \mathbf{s} = v * \mathbf{x} :\ \epsilon\} \Rightarrow$
$\{\mathbf{x} :\ \epsilon\}$
$\{all\_ST\ \epsilon\}$
}
$\{all\_ST\ \epsilon\}$