IT 10 052
Examensarbete 30 hp
Oktober 2010

Verifying Psi-calculi

Johannes Åman Pohjola

Institutionen för informationsteknologi
Department of Information Technology

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Verifying Psi-calculi

Johannes Åman Pohjola

Psi-calculi are mobile process calculi, parametrised with arbitrary nominal datatypes representing data, communication channels, assertions and conditions, as well as morphisms over those datatypes. The framework for psi-calculi has been formalised in the interactive theorem prover Isabelle, along with both strong and weak bisimulation.

This master's thesis project presents a tool for formally verifying that psi-calculus candidates are well-defined within the Isabelle/HOL-Nominal framework. It employs custom-made, heuristic proof tactics that discharge as many proof obligations as possible automatically, and passes any remaining proof obligations back to the user, who must supply manual proofs. The implementation of the tool as well as the proof strategies employed are described. The tool is applied to verify encodings of both the monadic and polyadic variants of the pi-calculus, as well as the pi-F calculus.

Handledare: Jesper Bengtson
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Anders Jansson
IT 10 052
Tryckt av: Reprocentralen ITC

Svensk sammanfattning

En gång i tiden var datorprogram sekventiella - de bestod av instruktioner staplade efter varandra, som datorn utförde i en sekvens. Idag är de flesta datorprogram flertrådiga, vilket innebär att de består av flera separata instruktionssekvenser, så kallade trådar, som datorn utför parallellt. Arbetet mellan dessa måste synkroniseras, och de måste samsas om gemensamma resurser, till exempel minnesblock.

En gång i tiden körde datorer bara ett program åt gången. Idag har de flesta persondatorer flera processorer, och olika program kan köras samtidigt på olika processorer. Även en enskild processors klockcykler delas mellan flera program, genom så kallad multikörning. Arbetet mellan dessa program måste synkroniseras, och de måste samsas om gemensamma systemresurser.

En gång i tiden körde datorer sina program i ensamhet, utan någon interaktion med andra datorer. Idag går det knappt att föreställa sig en persondator utan internetuppkoppling.

Att förstå ett modernt datorsystem handlar i allt högre grad om att förstå samspelet mellan dess olika delsystem, men systemen är ofta för stora och komplexa för att kunna analyseras i sin helhet. Abstraktioner måste göras.

En processkalkyl är en matematisk modell som beskriver parallella system algebraiskt. Systemen representeras av så kallade agenter, och interaktionen mellan agenter representeras av kommunikation. Hur agenter kan bete sig och interagera definieras formellt som algebraiska räkneregler, och på så sätt kan man resonera formellt om agenters egenskaper och relationerna mellan dem.

Psi-kalkylerna är en familj av processkalkyler som definieras av ett antal parametrar. Genom att ge parametrarna olika värden kan man erhålla olika psi-kalkyler, som vi kallar instanser, specialanpassade för att beskriva olika tillämpningar. Parametrarna har valts så att många generella egenskaper kan bevisas för psi-familjen som helhet, snarare än var för sig för varje instans. Att visa dessa egenskaper för en kalkyl som definierats utanför psi-ramverket brukar innebära långa och arbetssamma fallbevis, med många fallgropar för intuitionen.

Parametrarna som definierar en instans får inte väljas hur som helst, utan vissa krav på dem måste vara uppfyllda för att en instans ska vara korrekt. Bevisen som måste göras för att visa att dessa krav är uppfyllda är inte principiellt svåra, men innehåller en del teknikaliteter som gör att de ändå blir omständliga.

Det här arbetet presenterar ett verktyg för att automatisera dessa bevis i en teorembevisare, ett datorprogram som kan bevisa matematiska påståenden. Eftersom datorprogram saknar mänsklig intuition är det mycket svårare att konstruera ett bevis som övertygar en dator än ett som övertygar en människa, då inga detaljer kan lämnas åt fantasin. De mest triviala detaljer i ett bevis, sådant som vi människor inte ens tycker är värt att nämna, kan kräva förvånansvärt stora ansträngningar för att en teorembevisare ska låta sig övertygas.

En stor fördel med att använda teorembevisare är att när man väl får igenom sitt bevis, kan man vara helt förvissad om att det faktiskt är korrekt och att man inte missat några detaljer. Det blir också enklare att upptäcka felaktigheter i ett bevis om man tvingas förklara det tillräckligt detaljerat för att en teorembevisare ska förstå det.

Genom att automatisera bevisprocessen kan vi både äta kakan och behålla den - vi besparas den extra möda det innebär att förklara bevisen för en teorembevisare, utan att gå miste om förvissningen om att våra resultat är korrekta.

Bevismetoderna som verktyget använder är inte garanterade att alltid lyckas - om vi visste på förhand att det fanns en bevisstrategi som alltid fungerade, skulle bevisen inte behöva göras. Antalet möjliga sätt ställa in parametrarna när man definierar en psi-instans är oändliga, och att hitta en generell bevisstrategi som verifierar samtliga dessa är inte realistiskt. Vad verktyget däremot kan göra, är att fånga och generalisera de gemensamma mönster man kan skönja i bevisen för de instanser som förekommer i praktiken. Och vad verktyget kan garantera, är att om det hittar ett bevis, så är det beviset också korrekt.

Contents

Chapter 1

Background

1.1 Introduction

A process calculus is a mathematical tool that allows algebraic reasoning about concurrent computer systems. Processes are represented by mathematical constructs called agents, and interaction between independent agents is represented by communication (as opposed to, say, reading and writing shared memory). This technique admits formal reasoning about the properties of agents, and equivalences between them - for an example, bisimulation is an equivalence relation between agents that can imitate the behaviour of each other.

The pi-calculus [17] is a mobile process calculus, meaning that it can describe concurrent computations whose configuration changes during execution. The pi-calculus in itself is rather bare-boned, as it only features a single type of data: communication channels. Many practical applications use various extensions of the pi-calculus that sacrifice simplicity for modelling convenience. Examples include the spi-calculus [3] which focuses on cryptographic applications, and the applied pi-calculus [2] which features arbitrary datatypes.

Defining a new process calculus requires a substantial amount of theoretical groundwork, and the proofs involved are often gruesome and error-prone to work with.

The psi-calculi [9] is a family of mobile process calculi, a member of which we call an instance. An instance of psi-calculi is defined by instantiating a small number of parameters. As long as the instantiation of these parameters satifies a modest set of requirements, meta-theoretical reasoning such as bisimulation proofs can be done for the family of psi-calculi as a whole rather than for specific instances. Thus, when designing a new process calculus, defining it as a psi-instance spares the designer the burden of having to do the meta-theoretical proofs from scratch - a process that would typically involve long and tedious case analysis, with many tempting pitfalls for the human intuition.

Defining a new process calculus as a psi-instance ensures that the calculus will be correct[1]. This can be asserted since the framework for psi-calculi has been formalised in a theorem prover [10], a computer program that can prove mathematical theorems. However, in order to assert with confidence that we have inherited this correctness when defining an instance, we need to prove formally that the instance is well-defined.

One of the properties of a well-defined instance is that the substitution functions must be well-behaved - that is, they must satisfy the so-called substitution lemmas. In early experiments with formalising the pi-calculus as an instance, these proofs were found to be, while occasionally long and tedious, intuitively simple - even simple enough to admit proof automation.

The aim of this project is to design a tool that simplifies the formal verification of psi-instances. As input, the user provides the functions and nominal datatypes that define a psi-instance. The tool applies automatic proof procedures that discharge as many proof obligations as possible, and passes control to the user for those obligations where the automatic proof procedures fail. Moreover, this project aims to present a collection of example instances that should serve to illustrate both the expressiveness of psi-calculi and the workings of the above-mentioned tool.

1.2 Isabelle

Isabelle [21] is a generic interactive theorem prover. It is developed at the University of Cambridge, the Technische Universität München, and the Université Paris-Sud. It is implemented in Standard ML.

At the core of Isabelle lies *terms*, which are essentially typed λ-terms. A *theorem* is a term whose type is that of a proposition, and that can be

[1]Correct in the sense that the proofs outlined in [10] hold. That is, that bisimilarity is preserved by the operators in the expected way, and also satisfies the expected structural algebraic laws.

constructed only by applying a small set of primitives that correspond to the deduction rules of Isabelle's meta logic.

A proposition P is typically proven by means of first constructing the theorem $P \Rightarrow P$. To this tautology, *tactics* are applied that attempt to discharge the assumption P. On the implementation level, a tactic is a function with type `thm -> thm seq`[2] Intuitively, a tactic attempts to refine a theorem in some way and returns a sequence of all the refinements it can produce. In particular, the empty sequence is returned by a tactic that fails.

On top of this core, Isabelle branches out into many different layers of libraries, tactics and syntactic sugar, making it a very rich and workable framework for many applications of theorem proving. Definitions and theorems are collected in `.thy` files, *theories*, which may depend on other theories.

One extension to the Isabelle core that is frequently used in the Isabelle formalisation of psi-calculi is the sectioning concept known as *locales* [6]. A locale has a set of fixed assumptions, and theorems proved within the locale depend on those assumptions. For an example, there might be a locale for semigroups, which contains theorems about any set/operator pair satisfying the semigroup axioms. If we are working with the group $(\mathbb{N}, +)$, and want to use these theorems, we perform a *locale interpretation* - we prove that $(\mathbb{N}, +)$ is a semigroup, and the locale package automatically instantiates all the general theorems in the semigroup locale to concrete theorems about $(\mathbb{N}, +)$.

The meta logic is used to encode object logics such as First-order logic (FOL), Higher-order logic (HOL) and Zermelo-Fraenkel set theory (ZFC). The formalisation of psi-calculi is built on the HOL/Nominal object logic.

1.3 Nominal logic

A bound name n is a name occuring inside a structure that binds n. For an example, in the proposition $\forall n.n \wedge m$, n is considered a bound name because it occurs within a $\forall$-quantification over n. However, m, is not bound by any structure, and is considered *free* in the same expression.

A problem that is frequently glossed over when doing traditional pen-and-paper proofs, yet often becomes an issue when attempting to explain a proof to a computer, is the treatment of bound names. For an example, if

[2]Tactics return a lazily evaluated sequence of theorems. For details, see [28, pp. 34-35]

a bound name is caused to collide with a free name as the result of a name substitution, the free name might be accidentally captured by a binder.

The key observation is that the choice of bound names ultimately does not matter. For an example, the theorems $\forall ab.a+b = b+a$ and $\forall xy.x+y = y+x$ are clearly the same theorem, up to the choice of bound names, and it is desirable to be able to reason about them as though they were one and the same. We say that theorems such as these are *α-equivalent*.

Informal proofs often use the Barendregt variable convention [7], that all bound names are unique. Unfortunately, this turns out to be inapplicable in the general case [24]. Hence, more precise tools are required if we want to treat binders and α-equivalence formally.

The formalisation of psi-calculi in Isabelle uses nominal logic [22], a logical framework designed to admit a formal treatment of binders and α-equivalence. This is a whole research area in itself, but for our purposes, the following simplified treatment will suffice:

There is a countable set of *names* $\mathcal{N}$. $a, b, c, ..., x, y, z$ range over $\mathcal{N}$. A *name swapping* $(a\ b) \cdot c$ is defined as follows:

$$(a\ b) \cdot c = \begin{cases} a & \text{if } c = b \\ b & \text{if } c = a \\ c & \text{otherwise} \end{cases}$$

A sequence of name swappings is called a *permutation*, and we shall use $p \cdot c$ as a short-hand for $(a_1\ b_1) \cdot ... \cdot (a_n\ b_n) \cdot c$, where $p = [(a_1\ b_1), ..., (a_n\ b_n)]$.

A *nominal datatype* is a datatype equipped with such a name swapping function. Intuitively, $p \cdot X$ applies p to all names in X, where X is an element of some nominal datatype.

Definition The support $n(X)$ of X is the least set of names such that $(a\ b) \cdot X = X$ for all $a, b \notin n(X)$.

Definition a is fresh for X, written $a \sharp X$, iff $a \notin n(X)$.

Intuitively, the support of X is the set of free names in X. Any name that doesn't occur free in X is fresh for X. We will occasionally use the notation *supp*(X) to mean $n(X)$, in correspondence to the notation used in Isabelle.

The object logic HOL-Nominal [23] implements nominal logic in Isabelle, and provides the necessary infrastructure for reasoning about nominal datatypes. In particular, whenever a nominal datatype is defined, the HOL-Nominal package automatically provides a name swapping function and derives many useful theorems regarding permutation, support and freshness about the datatype "for free".

1.4 What is a Psi-instance?

A psi-calculus is defined by instantiating three nominal datatypes (see 1.3):

$\mathbf{T}$	(data) terms
$\mathbf{C}$	conditions
$\mathbf{A}$	assertions

three substitution functions:

$S_T : \mathbf{T} \times seq(\mathcal{N}) \times seq(\mathbf{T}) \rightarrow \mathbf{T}$	term substitution
$S_C : \mathbf{C} \times seq(\mathcal{N}) \times seq(\mathbf{T}) \rightarrow \mathbf{C}$	condition substitution
$S_A : \mathbf{A} \times seq(\mathcal{N}) \times seq(\mathbf{T}) \rightarrow \mathbf{A}$	assertion substitution

and four morphisms:

$\dot{\leftrightarrow} : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{C}$	channel equivalence
$\otimes : \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$	composition
$\mathbf{1} : \mathbf{A}$	unit
$\vdash \subseteq \mathbf{A} \times \mathbf{C}$	entailment

M, N, L, T range over $\mathbf{T}$, φ ranges over $\mathbf{C}$ and Ψ ranges over $\mathbf{A}$. For the substitution functions, we will use the notation $M[\widetilde{x} := \widetilde{T}]$ to mean $S_T(M, \widetilde{x}, \widetilde{T})$, and overload it so that we can analogously use $\varphi[\widetilde{x} := \widetilde{T}]$ and $\Psi[\widetilde{x} := \widetilde{T}]$ for condition and assertion substitution, respectively.

Terms are used to represent data, including communication channels. The channel equivalence condition $\dot{\leftrightarrow}$ is used to indicate that two terms represent the same communication channel, and is a precondition for all input, output and communication actions in the calculus. Assertions represent information about the environments in which agents act. The $\otimes$ operator can be thought of as the conjunction of the information in two assertions. $\mathbf{1}$ is the assertion that carries the least possible information. Entailment $\vdash$ relates assertions to conditions. Intuitively, $\Psi \vdash \varphi$ means that φ follows from the information in Ψ. Assertions that entail the same conditions are considered equivalent:

Definition Two assertions are *statically equivalent*, written $\Psi \simeq \Psi'$, iff $\forall \varphi. \Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$

The parameters described above are subject to the following requisites:

Term equivariance:	$p \cdot M[\widetilde{x} := \widetilde{M}] = (p \cdot M)[(p \cdot \widetilde{x}) := (p \cdot \widetilde{M})]$
Term freshness:	$\widetilde{x} \subseteq n(M) \wedge a\sharp M[\widetilde{x} := \widetilde{N}] \Rightarrow a\sharp\widetilde{N}$
Term α-equivalence:	$p \subseteq \widetilde{x} \times (p \cdot \widetilde{x}) \wedge p \cdot \widetilde{x}\sharp M \wedge \mathit{distinct}(p)$ $\Longrightarrow$ $M[\widetilde{x} := \widetilde{N}] = (p \cdot M)[(p \cdot \widetilde{x}) := \widetilde{N}]$

Analogous equivariance, freshness and α-equivalence requisites for conditions and assertions.

Channel equivariance:	$p \cdot (M \dot{\leftrightarrow} N) = (p \cdot M) \dot{\leftrightarrow} (p \cdot N)$
Composition equivariance:	$p \cdot (\Psi \otimes \Psi') = (p \cdot \Psi) \otimes (p \cdot \Psi')$
Unit equivariance:	$p \cdot \mathbf{1} = \mathbf{1}$
Entailment equivariance:	$p \cdot (\Psi \vdash \varphi) = (p \cdot \Psi) \vdash (p \cdot \varphi)$

Channel symmetry:	$\Psi \vdash M \dot{\leftrightarrow} N \Rightarrow \Psi \vdash N \dot{\leftrightarrow} M$
Channel transitivity:	$\Psi \vdash M \dot{\leftrightarrow} N \wedge \Psi \vdash N \dot{\leftrightarrow} L \Rightarrow \Psi \vdash M \dot{\leftrightarrow} L$
Composition:	$\Psi \simeq \Psi' \Rightarrow \Psi \otimes \Psi'' \simeq \Psi' \otimes \Psi''$
Identity:	$\Psi \otimes \mathbf{1} \simeq \Psi$
Associativity:	$(\Psi \otimes \Psi') \otimes \Psi'' \simeq \Psi \otimes (\Psi' \otimes \Psi'')$
Commutativity:	$\Psi \otimes \Psi' \simeq \Psi' \otimes \Psi$

Here, $\mathit{distinct}(p)$ is a predicate that is true if and only if no name occurs twice in p.

The equivariance, freshness and α-equivalence properties of terms, assertions and substitutions will be collectively referred to as the *substitution lemmas*. A datatype with a substitution function as defined above, for which the substitution lemmas hold, will be referred to as a *substitution type*.

Finally, note that this presentation differs somewhat from earlier presentations. Since the publication of [9] and [10], most of the requisites on the substitution functions presented therein were discovered to be superfluous, and have been removed. For a more recent presentation that reflects this, see [15, pp. 40-43]. Also, this presentation puts greater emphasis on the properties of the substitution functions, since they play a central part in this project.

Chapter 2

Problem description

2.1 Problem formulation

2.1.1 Learning the preliminaries

The first step of the project is to learn about Isabelle and psi-calculi. This includes learning:

- The framework for psi-calculi
- Isabelle at the user level: how to conduct proofs
- Isabelle at the implementation level
- Relevant parts of the Isabelle formalisation of psi-calculi. In particular, how the proofs when verifying a psi-instance are structured.

2.1.2 The tool

The user, having a candidate psi-instance I in need of formal verification, provides an encoding of $e(I)$ of I in Isabelle/HOL. The tool takes $e(I)$ as input, and does the following:

- Generate the proof obligations necessary to show that $e(I)$ is a psi-instance (see section 1.4)
- Discharge as many of these proof obligations as possible using automated proof procedures.

- When the automated proof procedures cannot discharge a proof obligation, it is passed to the user, who will have to produce a manual proof.

2.1.3 The examples

Along with the tool, a collection of example instances along with their verification should be presented. These should serve to illustrate both the expressiveness of the psi-calculus and the workings of the tool.

2.2 Problem structure

Originally, the intention was to develop the tool hand in hand with example instances. The idea was to start from a trivial instance and develop the tool to the point where this instance could be verified automatically, and from that point, proceed to gradually more interesting example instances, making the tool more sophisticated in order to accomodate their verification.

However, upon learning more about Isabelle programming, it gradually became clear that the sheer amount of infrastructure needed to verify any instance would make this approach infeasible. A significant amount of project time would have to be spent developing the tool before any meaningful instance could be verified, thus delegating the study of example instances to the tail-end of the project's time span.

Work on implementing the tool naturally fell into two distinct categories: the development of proof automation procedures, and the development of infrastructure. The infrastructure includes things such as construction of proof obligations from psi-instance parameters, parsing of user input, deploying automatic proof procedures, passing unproven proof obligations to the user, and so on.

Additionally, the work on proof automation fell into three categories:

- Understanding how the proofs should be structured - designing heuristics tailored for solving particular proof obligations
- Implementing those heuristics as tactics
- Infrastructure for passing the right lemmas and simplification rules to the tactics.

For the work on proof obligations, particular emphasis was put on the substitution lemmas, for two reasons. First, the substitution lemmas are low-level details that a user ideally shouldn't have to bother with. Second, the proofs of the substitution lemmas seems to follow a common proof structure that is fairly independent of the particular details of the instance being studied, thus making them more amenable to proof automation.

Chapter 3

Method

3.1 Learning

When this project started, the only material on psi-calculi in existence was two conference articles, one describing the psi-calculus framework [9] and the other describing its Isabelle formalisation [10]. Hence, learning was mainly a matter of reading those papers. Having the luxury of a one-minute walking distance from my workspace to the offices of all four authors gave me the opportunity to ask questions whenever anything was unclear.

For learning Isabelle on the user level, there exists quite a rich flora of manuals, tutorials and exercises to learn from. Among these, I have been using the slides from an Isabelle/HOL course by Nipkow [20], as well as the various tutorials that are available [21] [19] [5]. Additionally, reading the theory files in the Isabelle library has been very helpful for understanding how things are defined and what theorems are available.

To get some experience working on the user level of Isabelle, a smaller project was undertaken, which involved proving several correctness properties of a recursive solution to the Towers of Hanoi problem.

There are far fewer resources available for getting familiar with Isabelle programming. Some parts of the Isabelle source code are well documented - in particular, the older modules written by Paulson feature quite extensive internal documentation, which helps tremendously. Other parts of the code, unfortunately, have almost no internal documentation, which leaves only two options: ask the developers, or study the source code.

There are two manuals currently in development, whose drafts help shed light on the situation: the Isabelle/Isar Implementation manual [28], and the Isabelle Cookbook [11]. The former provides a good reference on how the various modules of Isabelle are implemented, while the latter is a practically oriented, example-driven tutorial on how to write ML code for Isabelle.

Guided by these two manuals, I learned Isabelle programming primarily by means of experimentation. The interactive development environment of Proof General is an excellent platform for such experimentation. Blocks of ML code can be embedded into Isabelle theories and then evaluated on the fly, and their effect on the user-level can be easily explored. Good starting points for experimenation were provided by the many code examples in the Isabelle Cookbook.

Also, a visit to the Isabelle development team in München was carried out by Palle Raabjerg and myself, allowing us to learn directly from the developers. This was particularly helpful for learning about features that were not covered in the manuals.

3.2 Proof automation

The formalisation of the pi-calculus as a psi-instance was studied in detail, and proof ideas were largely adapted from there. To get a deeper understanding of the proof structure and make sure that the proof methods would generalise well to other instances, pen-and-paper proofs were sketched for instances with other datatypes but similar substitution functions. The primary proof method used was induction over the nominal datatypes.

The Isabelle codebase features a rich library of tactics and tactic combinators with which to implement proof methods. In particular, Isabelle has a powerful simplifier that performs automatic term rewriting, significantly decreasing the amount of manual work required by the user. The simplifier takes a *simpset* as input. For our purposes, the simpset can be thought of as a set of term rewriting rules. These can take the form of theorems, or so-called *simprocs*, rewriting rules that are only triggered by the occurence of some specific pattern in the goal state. Some care must be taken when choosing the simpset. If the simpset is too small, the proof will not go through. If the simpset is too large, a wrong turn at some point might lead the simplifier into a dead end or a loop. To determine which rules should go into the simpsets, traces are performed on the simplifier to determine what term rewriting rules are actually used during the proofs.

For some of the more difficult proofs, induction followed by a call to the simplifier is insufficient, and more sophisticated structure has to be imposed on the proofs. Hierarchies of tactics, each responsible for a small part of the proof, are constructed and linked together, in a way inspired by the proof structuring mechanisms of Isar.

3.3 Infrastructure

To make the tool easily usable from the user level of Isabelle, it is necessary to introduce new Isar commands. These are implemented as transitions from a local theory (which can be either a theory or a locale) to a proof state the user can manipulate. The proof state returned is the first proof obligation for which automatic proof procedures fail, or a trivial proof state with no subgoals if all proof obligations are discharged automatically. Parsing of parameters for these commands is implemented using the many parsers and parser combinators available in the Isabelle library, particularly the `OuterParse` and `Syntax` modules.

Since the tool employs heuristic proof tactics which are not guaranteed to work, a data structure to describe the result of a proof attempt, rather than a proof, is necessary. Proof procedures are wrapped in functions that returned an `attempt_result`, a data structure containing:

- in the event of a successful proof, a new local theory in which the new theorem has been registered

- in the event of a failed proof, a term representing the proof obligation, and a name under which to register it in the local theory once a successful proof is made.

For the substitution lemmas, the order of the proofs is important, since they use induction over the nominal datatypes. For an example, if conditions contain assertions, the proofs of the substitution lemmas for conditions would need to apply the corresponding lemmas about assertions in the inductive step. Given the nominal datatypes which describe a psi instance, a graph describing the relationship between them is constructed, and this graph is sorted topologically. The proofs are then attempted in this topological order.

3.4 Example instances

The existing publications on psi-calculi, such as [9] and [15], already feature various examples of encodings of other calculi as psi-instances. However, these examples are not formalised in a theorem prover.

Examples are borrowed from these publications, and they are encoded in Isabelle, using **nominal_datatype** definitions to encode the nominal datatypes, and **fun** and **nominal_primrec** declarations to encode the morphisms and substitution functions.

3.5 Division of labour

Throughout most of this project, it has been a collaborative effort with Palle Raabjerg. Early on in the project's lifespan, he was responsible for developing tactics, while I was responsible for developing infrastructure. Some of his main contributions include:

- The early work on figuring out how to write tactics. In particular, figuring out how to apply nominal induction on the ML-level.
- Much of the work on figuring out how to do proofs involving existential quantification (corresponding to Isar's `obtain` command) on the ML-level, as well as adapting these to the proof of the freshness substitution lemma.

Specifically, my own contributions are:

- Design and implementation of the infrastructure, as described in section 4.1.
- Design and implementation of the example instances described in section 4.3.
- As for the proof methods, their formulation in section 4.2 is mine, although the methods are inspired by Bengtson's Isabelle proofs for the pi-calculus instance. I've also made the implementation of the proofs for the $\sharp^*$ properties described in section 4.2.2, as well as many of the low-level procedures of the freshness proofs described in section 4.2.5. Otherwise, the implementation of the proof methods is due to Palle Raabjerg, with some debugging and refactoring by me.

Chapter 4

Results

The project's results are presented under three main categories: program structure, proof methods and example instances.

Section 4.1 describes how the implementation of the tool is structured, what its main components are, how they work and how they relate to each other.

Section 4.2 describes the proof strategies employed by the tool and their implementation as Isabelle tactics. In order to avoid bogging down the presentation with unnecessary details, some abstractions have been made. For any substitution $X[\widetilde{x} := \widetilde{M}]$, $\widetilde{x}$ and $\widetilde{M}$ must be of equal length for the substitution lemmas to hold, with the exception of equivariance. This requirement is not treated explicitly. Also, the notation $\widetilde{X}$ for sequences is overloaded to also denote the set $\{X.X \in \widetilde{X}\}$ of all elements in the sequence.

Finally, section 4.3 encodes various calculi as psi-instances in Isabelle, and describes how they can be verified with the aid of the tool.

4.1 Program structure

4.1.1 User level structure

The tool is built on top of a user-level infrastructure designed for manual verification of psi-instances. This infrastructure consist of the *nameSubst*, *substType* and *assertionAux* locales, all of them originally designed by Bengtson [8]. *nameSubst* is an adapted version of a locale used in his proof that the pi-calculus is a psi-instance. *substType* and *assertionAux* are part of his Isabelle formalisation of psi-calculi.

A psi-instance can be verified manually in Isabelle by performing the following steps:

1. Define **T**, **C** and **A** as nominal datatypes.
2. Interpret the *nameSubst* locale with **T** as a parameter to obtain the functions $substName$ and $nsCase$, auxiliary functions for defining substitution functions.
3. With the help of $substName$, define the substitution functions S_T, S_C and S_A.
4. Define the morphisms $\dot{\leftrightarrow}$, $\otimes$, $\mathbf{1}$ and $\vdash$.
5. Prove the substitution lemmas for **T**, **C** and **A**.
6. Interpret the *substType* locale for **T**, **C** and **A**, respectively.
7. Prove equivariance properties for $\dot{\leftrightarrow}$, $\otimes$, $\mathbf{1}$ and $\vdash$.
8. Interpret the *assertionAux* locale to obtain the definition of static equivalence.
9. Prove the remaining requisites on a psi-instance.

The role of this tool is to perform steps 5-9 of this process, or as much thereof as possible, automatically.

The *nameSubst* locale

The *nameSubst* locale takes only one parameter, the type **B**, and provides the following basic building blocks for psi-instances:

- A substitution function $substName : \mathcal{N} \times seq(\mathcal{N}) \times seq(\mathbf{B}) \to \mathbf{B} \cup \mathcal{N}$ similar to those defined in section 1.4:
  $$substName(a, \widetilde{x}, \widetilde{B}) = \begin{cases} \widetilde{B}_i & \text{if } \widetilde{x}_i = a \text{ and } \forall j < i.\widetilde{x}_i \neq a \\ a & \text{otherwise} \end{cases}$$
  This function is intended to handle the base case of names when defining the substitution functions of a psi-instance. The locale includes all the necessary theorems to show that $\mathcal{N}$ is a substitution type under this substitution function.

- A higher-order case split function $nsCase : \mathcal{N} \cup \mathbf{B} \times (\mathcal{N} \rightarrow \text{'a}) \times (\mathbf{B} \rightarrow \text{'a}) \rightarrow \text{'a}$, defined as follows:

  $$nsCase(a, f, g) = \begin{cases} f(a) & \text{if } a \in \mathcal{N} \\ g(b) & \text{otherwise} \end{cases}$$

  This rather convoluted case split function is necessary becase Isabelle's default case statement doesn't admit non-equality types, and nominal datatypes are not equality types in the general case. Theorems stating the usual freshness, support and equivariance properties for *nsCase* are supplied by the locale. We also introduce the syntax

  $$cases\ a\ of\ names \Rightarrow f \mid terms \Rightarrow g$$

  to mean $nsCase(a, f, g)$

The *substType* locale

The *substType* locale defines substitution types. When verifying a psi-instance, it should be interpreted thrice: once each for terms, condition and assertions.

The *assertionAux* locale

The *assertionAux* locale takes as parameters equivariance theorems for four morphisms satisfying the signature of the four morphisms that define a psi-instance. Along with many useful theorems, a definition of static equivalence is derived, which is needed to state the composition, identity, associativity and commutativity requisites of a psi-instance.

4.1.2 Top-level

The tool can be interfaced from the Isabelle user level by means of two new Isar commands:

- **psi_instance**. As parameters, it takes the nominal datatypes and morphisms which define a psi-instance, and attempts to verify that the instance is well-defined. It puts the user in a proof context for the first pending proof obligation which it couldn't prove, or a trivial proof context if there are no pending proof obligations.

- **continue_psi** Takes the name of a psi-instance as a parameter. After a pending proof obligation has been proven by the user, the **continue_psi** command picks up where a previous pass of **psi_instance** or **continue_psi** left off. It attempts to discharge all remaining proof obligations, and then puts the user in a proof context exactly as for the **psi_instance** command.

  If the message `No subgoals!` appears when running the **continue_psi** command, the psi-instance is fully verified.

It should be noted that this structure, where multiple passes of a command are sometimes required, is somewhat unconventional. The established convention for Isar commands that might generate several proof obligations for the user is to present them as subgoals of a single single proof state, thus requiring only one pass of the command. Here, we instead present the unresolved proof obligations one at a time. This design decision is motivated by the fact that our proof obligations frequently depend on each other. For an example, if terms occur within assertions, the proofs of the substitution lemmas for assertions will depend on the substitution lemmas for terms. As a second example, the definition of static equivalence cannot be obtained until all morphisms have been proven to be equivariant, meaning that some of the proof obligations cannot even be stated until all equivariance properties are proven. This makes it convenient to handle the proofs sequentially.

The data flow of these commands are described in figures 4.1 and 4.2, respectively. The modules that these commands are built from are described in sections 4.1.3 through 4.1.8.

4.1.3 Data structures

This section describes some of the key data structures used for implementing the infrastructure.

- `auxiliary`, defined as follows:

  ```
  type auxiliary = {typ: typ, subst: Term.term,
                    simps: thm list, invariant: Term.term}
  ```

  This structure is used for two similar, and sometimes interchangeable, purposes: recording auxiliary information about a datatype `typ`, and

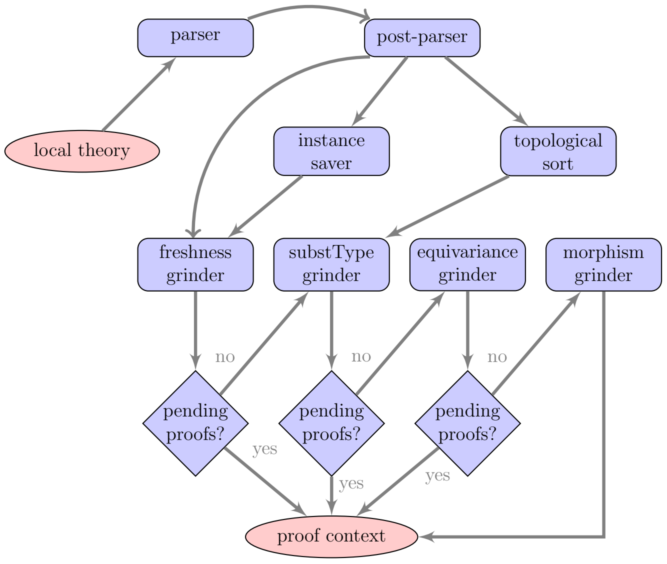

Figure 4.1: Data flow of the psi-instance command

recording information about an auxiliary datatype `typ`. In both cases, its substitution function is `subst`. `simps` contains a list of simplification rules relevant to the data structure. The `invariant` field is intended to record a datastructure invariant, a possible extension of the framework. However, this has not been implemented, so this field is currently not used.

- The `psi_instance` data structure records all the necessary parameters which define a candidate psi-instance, as well as various auxiliary information about it.

```
type psi_instance =
  {name: string,
   terms: typ,
   terms_aux: auxiliary,
   conditions: typ,
   conditions_aux: auxiliary,
   assertions: typ,
   assertions_aux: auxiliary,
```

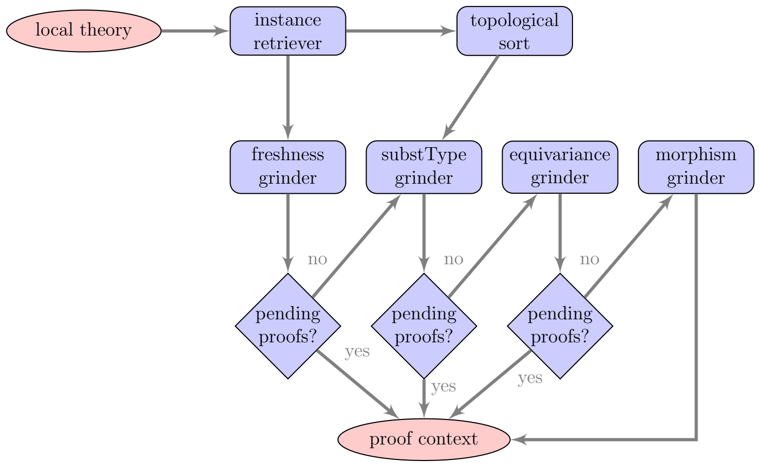

Figure 4.2: Data flow of the continue-psi command

```
    termsubst: Term.term,
    condsubst: Term.term,
    assertsubst: Term.term,
    chaneq: Term.term,
    comp: Term.term,
    bottom: Term.term,
    entailment: Term.term,
    auxiliaries: auxiliary list}
```

The `bottom` field contains the unit assertion **1**. The `auxiliaries` field is a list of substitution types which aren't parameters to the psi-instance in and of themselves, yet are used as auxiliary structures in the definitions of the actual parameters. Otherwise, the meaning of the fields are largely self-explanatory.

- The `attempt_result` data structure is used to describe the result of an attempted proof:

```
datatype attempt_result = Proven of local_theory
                        | Unproven of Term.term *
                                     Attrib.binding
```

An unsuccessful attempt to prove the proposition `P` in the environment `lthy` is described as `Unproven(P,b)`, where `b` is the name that should be given to the theorem once successfully proven.

A successful attempt to prove the same proposition is described by `Proven(lthy')`, where `lthy'` is `lthy` with the theorem `P` registered under the name `b`.

- It is also necessary to describe the result of grinders (see section 4.1.6), or in other words, the result of the sequential proof attempts of a list of propositions. This is implemented as a record type:

  ```
  local_theory * (Term.term * Attrib.binding) option
  ```

  A successful attempt to prove the propositions $P_1, ..., P_n$ in the environment `lthy` is described by `(lthy', NONE)`, where `lthy'` is `lthy` with $P_1, ..., P_n$ registered as theorems under appropriate names.

  An attempt to prove the same propositions which doesn't successfully prove P_i is described by `(lthy', SOME(P_i, b))`, where `lthy'` is `lthy` with $P_1, ..., P_{i-1}$ registered as theorems, and `b` is the name that should be given to P_i once successfully proven.

4.1.4 Parser

The parser is responsible for parsing user input, and accepts an instance as defined by the following grammar:

$$
\begin{array}{lcl}
\langle \mathit{instance} \rangle & \rightarrow & \text{“psi\_instance”} \ \langle \mathit{string} \rangle \ \text{“:”} \ \langle \mathit{items} \rangle \\
\\
\langle \mathit{items} \rangle & \rightarrow & \langle \mathit{item} \rangle \ \text{“|”} \ \langle \mathit{items} \rangle \\
 & | & \langle \mathit{item} \rangle \\
\\
\langle \mathit{item} \rangle & \rightarrow & \langle \mathit{typeparam} \rangle \ \text{“=”} \ \langle \mathit{typename} \rangle \\
 & | & \langle \mathit{typeparam} \rangle \ \text{“=”} \ \langle \mathit{typename} \rangle \ \text{“where”} \ \langle \mathit{options} \rangle \\
 & | & \langle \mathit{termparam} \rangle \ \text{“=”} \ \langle \mathit{termname} \rangle \\
 & | & \langle \mathit{termparam} \rangle \ \text{“=”} \ \langle \mathit{termname} \rangle \ \text{“where”} \ \langle \mathit{options} \rangle \\
 & | & \text{“auxilary”} \ \langle \mathit{typename} \rangle \ \text{“where”} \ \langle \mathit{options} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathit{typeparam} \rangle & \rightarrow & \text{“terms”} \mid \text{“conditions”} \mid \text{“assertions”} \\
\\
\langle \mathit{termparam} \rangle & \rightarrow & \text{“termsubst”} \mid \text{“condsubst”} \\
 & \mid & \text{“assertsubst”} \mid \text{“namecast”} \\
 & \mid & \text{“chaneq”} \mid \text{“comp”} \\
 & \mid & \text{“unit”} \mid \text{“entailment”} \\
\\
\langle \mathit{options} \rangle & \rightarrow & \langle \mathit{option} \rangle \text{ “,” } \langle \mathit{options} \rangle \\
 & \mid & \langle \mathit{option} \rangle \\
\\
\langle \mathit{option} \rangle & \rightarrow & \langle \mathit{invariant} \rangle \text{ “=” } \langle \mathit{predicate} \rangle \\
 & \mid & \langle \mathit{simps} \rangle \text{ “=” } \langle \mathit{theorems} \rangle \\
 & \mid & \langle \mathit{subst} \rangle \text{ “=” } \langle \mathit{termname} \rangle
\end{array}
$$

The nonterminals ⟨*typename*⟩ and ⟨*termname*⟩ are strings that name types or terms defined in the local theory, ⟨*precidate*⟩ is a string that can be parsed as a term of type *prop* in the local theory, and ⟨*theorems*⟩ is a string which defines a list of theorems according to the standard Isar syntax. A ⟨*string*⟩ is a regular string.

The output of the parser is a list of items, along with a string representing the name of the instance. This is passed along to the post-parser described in the next section.

4.1.5 Post-parser

The post-parser receives a list of items and an instance name from the parser, and attempts to construct a `psi_instance` from these. If successful, the `psi_instance` is passed on to the instance saver to record information about it in the local theory, and to the grinders that attempt to solve the proof obligations.

Some rudimentary sanity checks on the user input are performed in this step. For an example, if any of the psi-instance parameters are missing, the post-parser will raise an exception.

4.1.6 Grinders

A grinder is responsible for attempting the sequential verification of a list of proof obligations. The basic structure of a grinder is described in Figure 4.3.

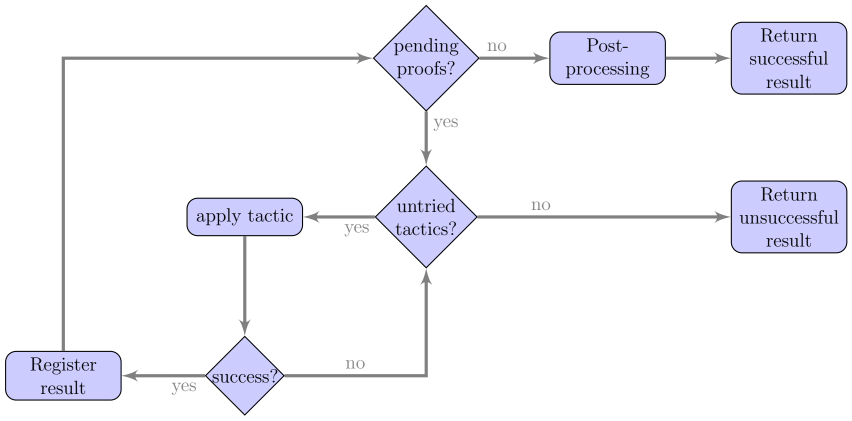

Figure 4.3: Structure of a grinder

For each of these proof obligations, the grinder first makes sure that it hasn't already been proved by a previous run of the grinder (or the user). If not, a sequence of tactics is applied that attempts to discharge it. If a successful proof attempt is made, this result is registered in the local theory, and the grinder proceeds to the next proof obligation.

If a pending proof obligation is encountered that cannot be proved by its tactics, the grinder stops there, and returns the first pending proof obligation (as described in 4.1.3). If no pending proof obligations were discovered, a post-processing step is applied to the local theory, the exact contents of which differs from grinder to grinder.

The different grinders featured in the implementation, as well as their responsibilities, are detailed below.

Freshness grinder

When a nominal datatype is defined, freshness lemmas that distribute the $\sharp$ operator over the datatype's components are automatically generated. However, the corresponding lemmas about the $\sharp^*$ operator (see section 4.2.2) are not. The freshness grinder is responsible for supplying these.

substType grinder

The substType grinder attempts to prove the substitution lemmas. As a post-processing step, it provides an interpretation of the *substType* locale for every substitution type.

Equivariance grinder

The equivariance grinder attempts to prove the equivariance properties about channel equivalence, entailment, composition and the unit element. As a post-processing step, it uses these to do an interpretation of the *assertionAux* locale. Among other things, this locale provides a notion of static equivalence, which is necessary for stating some of the subsequent proof obligations.

Morphism grinder

The morphism grinder is responsible for the remaining requisites on the psi-instance: channel symmetry, channel transitivity, composition, identity, associativity and commutativity. It does not feature any post-processing step.

4.1.7 Instance saver and retriever

The information recorded in a `psi_instance` (see section 4.1.3) structure needs to be stored in a way that allows it to be retrieved by future passes of the **continue_psi** command. Other than using references, which break Isabelle's undo mechanism, there is no simple way to preserve arbitrary ML data between commands in a local theory. The solution employed here is to encode it as Isabelle terms, which are saved in the local theory in a way analogous to the **definition** command of Isar.

At the beginning of every pass-through of **continue_psi**, these definitions are retrieved from the local theory and decoded, and the `psi_instance` is reconstructed.

4.1.8 Topological sort

As discussed in Section 3.3, the order in which the substitution lemmas are proved is important, since the proofs depend on each other.

To determine the proper order in which the substitution types should be treated, a dependency graph (V, E) is constructed, where:

- If X is a substitution type of the current psi-instance, $X \in V$.
- If Y occurs in the premises X's induction rule, where X is a substitution type in the current psi-instance, $Y \in X$
- $(X, Y) \in E$ iff Y occurs in the premises of X's induction rule.

This dependency graph is then sorted topologically, and the substitution types will be treated by the substType grinder in the obtained order.

4.2 Proof methods

4.2.1 Preconditions and a running example

To illustrate the idea behind the proof heuristics employed for the substitution lemmas, we will assume an arbitrary inductively defined nominal datatype Y, where y ranges over Y, without binders or mutual recursion:

$$\textbf{nominal\_datatype}\, Y = \begin{array}{lll} C_1\, E_{1,1} & \dots & E_{1,c_1} \\ \vdots & & \vdots \\ C_n\, E_{n,1} & \dots & E_{n,c_n} \end{array}$$

Given this definition, the nominal datatype package automatically provides freshness, support and permutation theorems which distribute onto the datatype components. For every constructur $C_i, 1 \leq i \leq n$ of the datatype, we have:

$$\begin{array}{ll} p \cdot (C_i\, e_{i,1} \dots e_{i,c_i}) = C_i\, (p \cdot e_{i,1}) \dots (p \cdot e_{i,c_i}) & \text{Y.perm} \\ x \sharp (C_i\, e_{i,1} \dots e_{i,c_i}) = (x \sharp e_{i,1}) \wedge \dots \wedge (x \sharp e_{i,c_i}) & \text{Y.fresh} \\ \mathit{supp}(C_i\, e_{i,1} \dots e_{i,c_i}) = (\mathit{supp}\, e_{i,1}) \cup \dots \cup (\mathit{supp}\, e_{i,c_i}) & \text{Y.supp} \end{array}$$

For each constructor C_i, let k_i be the number of elements e in $\{e_{i,1}, \dots, e_{i,c_i}\}$ such that $e \in Y$, and let $\langle y_{i,1}, \dots, y_{i,k_i} \rangle$ be the sequence of these elements, in order of their respective appearances in $\langle e_{i,1}, \dots, e_{i,c_i} \rangle$. The induction rule for Y, also generated automatically when defining the datatype, can then be stated as:

$$\dfrac{\dfrac{P(y_{1,1}) \quad ... \quad P(y_{1,k_1})}{P(C_1\ e_{1,1}\ ...\ e_{1,c_1})} \quad ... \quad \dfrac{P(y_{n,1}) \quad ... \quad P(y_{n,k_n})}{P(C_n\ e_{n,1}\ ...\ e_{n,c_n})}}{P(y)}\ \text{Y.induct}$$

Each component $E_{i,j}$ of Y, where $1 \leq i \leq n, 1 \leq j \leq c_i$, is assumed to be either:

- $\mathcal{N}$
- Y
- Another substitution type.
- A datatype that doesn't carry names. If $E_{i,j}$ doesn't carry names, we have $p \cdot e = e$, $a\sharp e = True$ and $supp(e) = \{\}$ for each $e \in E_{i,j}$. While such datatypes are not substitution types, we will use the notation $e[\widetilde{x} := \widetilde{T}]$ to mean e for convenience.

Finally, Y is assumed to have a substitution function which pushes the substitution onto its components, behaving like this:

$$\begin{array}{rcl}(C_1\ e_{1,1}\ ...\ e_{1,c_1})[\widetilde{x} := \widetilde{T}] & = & C_1\ (e_{1,1}[\widetilde{x} := \widetilde{T}])\ ...\ (e_{1,c_1}[\widetilde{x} := \widetilde{T}]) \\ & \vdots & \\ (C_n\ e_{n,1}\ ...\ e_{n,c_n})[\widetilde{x} := \widetilde{T}] & = & C_n\ (e_{1,1}[\widetilde{x} := \widetilde{T}])\ ...\ (e_{n,c_n}[\widetilde{x} := \widetilde{T}])\end{array}$$

4.2.2 $\sharp^*$ properties

We often find it convenient to abuse the notation $\sharp$ by using sequences of names rather than single names - for an example, we write $\widetilde{x}\sharp M$ to mean that all elements of $\widetilde{x}$ are fresh for M. In Isabelle, this is captured by the operator $\sharp^*$, defined as follows:

$$\widetilde{x}\sharp^* M = \forall x \in \widetilde{x}. x\sharp M$$

For the proofs of the substitution lemmas, we need lemmas about $\sharp^*$ analogous to those about $\sharp$, that distribute $\sharp^*$ onto the datatype components. For each constructor C_i of Y, the $\sharp^*$ property analogous to $Y.fresh$ would be

$\widetilde{x}\sharp^*(C_i\ e_{i,1}\ ...\ e_{i,c_i}) = (\widetilde{x}\sharp^* e_{i,1}) \wedge ... \wedge (\widetilde{x}\sharp^* e_{i,c_i})$ YFresh

with one exception: for every component $e_{i,j}$ of C_i that is a name, the correponding conjunct on the right-hand side will be $e_{i,j}\sharp\widetilde{x}$ rather than $\widetilde{x}\sharp^* e_{i,j}$. This follows from $Y.fresh$ by the following argument:

$$\begin{aligned}
\widetilde{x}\sharp^*(C_i\ e_{i,1}\ ...\ e_{i,c_i}) &= \forall x \in \widetilde{x}.x\sharp(C_i\ e_{i,1}\ ...\ e_{i,c_i}) && (1)\\
&= \forall x \in \widetilde{x}.(x\sharp e_{i,1}) \wedge ... \wedge (x\sharp e_{i,c_i}) && (2)\\
&= (\forall x \in \widetilde{x}.x\sharp e_{i,1}) \wedge ... \wedge (\forall x \in \widetilde{x}.x\sharp e_{i,c_i}) && (3)\\
&= (\widetilde{x}\sharp^* e_{i,1}) \wedge ... \wedge (\widetilde{x}\sharp^* e_{i,c_i}) && (4)
\end{aligned}$$

Steps 1 and 4 are applications of the definition of $\sharp^*$, and step 2 uses $Y.fresh$. Step 3 follows from the fact that $\forall x.(Px \wedge Qx) = (\forall x.Px) \wedge (\forall x.Qx)$.

This proof strategy is implemented as a tactic which performs the following steps:

1. Until no longer possible, apply the definitions of $\sharp^*$ and $Y.fresh$ as substitution rules.

2. Apply the substitution rule *Complete_Lattice.ball_conj_distrib*, which states that $\forall x \in X.(Px \wedge Qx) = (\forall x \in X.Px) \wedge (\forall x \in X.Qx)$, until no longer possible.

3. Apply the theorem $(a = c) \wedge (b = d) \Rightarrow (a \wedge b) = (c \wedge d)$ as a backward rule until no longer possible.

After applying these steps to the goal $YFresh$, we will have c_i subgoals to discharge, where the j:th subgoal is $(\forall x \in \widetilde{x}.x\sharp e_{i,j}) = e_{i,j}\sharp\widetilde{x}$ if $e_{i,j}$ is a name, and $(\forall x \in \widetilde{x}.x\sharp e_{i,j}) = (\forall x \in \widetilde{x}.x\sharp e_{i,j})$ otherwise. To discharge each of them, the final step is:

4. For each subgoal, apply reflexivity if $e_{i,j}$ is not a name. Otherwise, apply the simplifier with a simpset containing:

   - The definition of freshness
   - *Y.supp*

- Two theorems about the support of names: $supp(x) = \{x\}$ and $\widetilde{x} = supp(\widetilde{x})$.
- The theorem $(\forall a \in A. a \notin \{x\}) = x \notin \{A\}$

In the following sections, we will continue abusing the notation $\widetilde{x}\sharp M$ to mean $\widetilde{x}\sharp^* M$. When referring to $Y.fresh$, we will interpret this as referring to either $Y.fresh$, $YFresh$, or both, depending on the context.

4.2.3 Substitution equivariance

To prove Y's equivariance property, stated

$$p \cdot (y[\widetilde{x} := \widetilde{T}]) = (p \cdot y)[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})]$$

we begin a backwards proof by applying $Y.induct$, instantiated with

$$P \triangleq \lambda y. p \cdot (y[\widetilde{x} := \widetilde{T}]) = (p \cdot y)[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})]$$

To complete the proof, we must discharge the hypotheses of this rule. For each constructor $C_i, 1 \leq i \leq n$, we can discharge the corresponding hypothesis, whose conclusion has LHS $= p \cdot ((C_i\, e_{i,1} \ldots e_{i,c_i})[\widetilde{x} := \widetilde{T}])$ and RHS $= (p \cdot (C_i\, e_{i,1} \ldots e_{i,c_i}))[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})]$ by rewriting the LHS as follows:

$$\begin{aligned}
\text{LHS} &= p \cdot ((C\, n \ldots m)[\widetilde{x} := \widetilde{T}]) && (1)\\
&= p \cdot (C_i\, (e_{i,1}[\widetilde{x} := \widetilde{T}]) \ldots (e_{i,c_i}[\widetilde{x} := \widetilde{T}])) && (2)\\
&= C_i\, (p \cdot (e_{i,1}[\widetilde{x} := \widetilde{T}])) \ldots (p \cdot (e_{i,c_i}[\widetilde{x} := \widetilde{T}])) && (3)\\
&= C_i\, ((p \cdot e_{i,1})[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})]) \ldots ((p \cdot e_{i,c_i})[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})]) && (4)\\
&= (C_i\, (p \cdot e_{i,1}) \ldots (p \cdot e_{i,c_i}))[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})] && (5)\\
&= (p \cdot (C_i\, e_{i,1} \ldots e_{i,c_i}))[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})] && (6)\\
&= \text{RHS} && (7)
\end{aligned}$$

Steps (1)-(3) and (5)-(7) are straightforward applications of the definition of Y's substitution function as well as $Y.perm$. Step (4), which distributes the application of p into the substitutions for each component, is justified as follows for each $e_{i,j}$:

- If $e_{i,j} \in Y$, by the induction hypothesis.
- If $e_{i,j} \in \mathcal{N}$, by the *nameSubstEqvt* theorem of the *nameSubst* locale (see section 4.1.1)
- If $e_{i,j}$ is an element of another substitution type, by the definition of a substitution type (see section 1.4).
- If $e_{i,j}$ does not carry names, then since $e_{i,j}[\widetilde{x} := \widetilde{T}] = e_{i,j}$ and $p \cdot e_{i,j} = e_{i,j}$:

$$\begin{aligned} p \cdot (e_{i,j}[\widetilde{x} := \widetilde{T}]) &= p \cdot e_{i,j} \\ &= e_{i,j} \\ &= e_{i,j}[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})] \\ &= (p \cdot e_{i,j})[(p \cdot \widetilde{x}) := (p \cdot \widetilde{T})] \end{aligned}$$

When implementing this proof strategy as an Isabelle tactic, most of the details in the above proof are handled automatically by the simplifier. The tactic works by first applying induction over Y, followed by applying the simplifier to each new subgoal. The simpset used contains the following theorems:

- Permutation rules for Y and all of its components.
- The definitions of the substitution functions for Y and all of its components that are substitution types or names.
- Equivariance theorems for the substitution functions of all of Y's components that are substitution types, other than Y.

4.2.4 α-equivalence

The proof of the α-equivalence property of Y, stated as:

$$\begin{aligned} &p \subseteq \widetilde{x} \times (p \cdot \widetilde{x}) \wedge p \cdot \widetilde{x} \sharp y \wedge distinct(p) \\ &\quad \Longrightarrow \\ &y[\widetilde{x} := \widetilde{T}] = (p \cdot y)[(p \cdot \widetilde{x}) := \widetilde{T}] \end{aligned}$$

is analogous to the equivariance proof detailed in section 4.2.3, in that it is done by applying induction over Y, followed by straight-forward term rewriting. The implementation as a tactic is also analogous, with the following simpset:

- Permutation rules for Y and all of its components.
- The definitions of the substitution functions for Y and all of its components that are substitution types or names.
- α-equivalence theorems for the substitution functions of all of Y's components that are substitution types, other than Y.
- $Y.fresh$

$Y.fresh$ is needed to justify α-conversion of Y's components. For an example, suppose $e_{i,j}$ is a component of Y. To justify the step

$$e_{i,j}[\widetilde{x} := \widetilde{T}] = (p \cdot e_{i,j})[(p \cdot \widetilde{x}) := \widetilde{T}]$$

the rule we wish to cite for justification has the form:

$$\frac{p \subseteq \widetilde{x} \times (p \cdot \widetilde{x}) \quad p \cdot \widetilde{x} \sharp y \quad distinct(p)}{e_{i,j}[\widetilde{x} := \widetilde{T}] = (p \cdot e_{i,j})[(p \cdot \widetilde{x}) := \widetilde{T}]}$$

and our induction hypothesis contains the following:

$$p \subseteq \widetilde{x} \times (p \cdot \widetilde{x}) \wedge p \cdot \widetilde{x} \sharp (C_i\, e_{i,1} \ldots e_{i,c_i}) \wedge distinct(p)$$

$Y.fresh$ allows us to obtain $p \cdot \widetilde{x} \sharp e_{i,j}$ from $p \cdot \widetilde{x} \sharp (C_i\, e_{i,1} \ldots e_{i,c_i})$ and thus makes the rule applicable.

4.2.5 Freshness

The freshness property of Y, stated as:

$$\widetilde{x} \subseteq supp(y) \wedge a \sharp y[\widetilde{x} := \widetilde{T}] \Rightarrow a \sharp \widetilde{T}$$

is the most difficult to prove of the substitution lemmas. To see why, we begin a backwards proof by applying $Y.induct$, generalising over $\widetilde{x}$ and $\widetilde{T}$. For each constructor C_i of Y, we will have to discharge a subgoal stating:

$$\widetilde{x} \subseteq supp(C_i\ e_{i,1}\ ...\ e_{i,c_i}) \wedge a\sharp(C_i\ e_{i,1}\ ...\ e_{i,c_i})[\widetilde{x} := \widetilde{T}] \Rightarrow a\sharp\widetilde{T}$$

For each component $e_{i,j}$ of C_i that is a substitution type, we have a rule stating that for all $\widetilde{z}, \widetilde{M}$,

$$\frac{\widetilde{z} \subseteq supp(e_{i,j}) \quad x\sharp e_{i,j}[\widetilde{z} := \widetilde{M}]}{x\sharp\widetilde{M}}\ fresh_{i,j}$$

In order to apply the above rule, with $\widetilde{z}$ and $\widetilde{M}$ instantiated to $\widetilde{x}$ and $\widetilde{T}$, we need to discharge the assumption $\widetilde{x} \subseteq supp(e_{i,j})$. However, this cannot be inferred from $\widetilde{x} \subseteq supp(C_i\ e_{i,1}\ ...\ e_{i,j})$, since $\widetilde{x}$ might carry names that are not in $e_{i,j}$, but in the support of other components of C_i. Thus, if we attempt to prove Y's freshness property using the same proof strategy as for equivariance and α-equivalence (see sections 4.2.3 and 4.2.4), we will be unable to apply the induction hypothesis, and more sophisticated techniques are needed. The key is to prune $\widetilde{x}$ of superfluous names:

Lemma 4.2.5.1 (Subst3Aux) *For all $\widetilde{x}, \widetilde{T}, e_{i,j}$, there are $\widetilde{y}$ and $\widetilde{U}$ such that $supp(e_{i,j}[\widetilde{x} := \widetilde{T}]) = supp(e_{i,j}[\widetilde{y} := \widetilde{U}])$, $\widetilde{y} = supp(e_{i,j}) \cap \widetilde{x}$ and $\widetilde{U} = \{\widetilde{T}_n.\widetilde{x}_n \in \widetilde{y}\}$.*

While proofs of this lemma have been found for some instances, none of the proof strategies employed seem to generalise well. Thus, it is currently not provided automatically by the tool, but must be proven manually by the user.

For each $e_{i,j}$, we use lemma 4.2.5.1 to obtain the sequences $\widetilde{y_{i,j}}$ and $\widetilde{U_{i,j}}$, and instantiate $\widetilde{z}$ and $\widetilde{M}$ with them in $fresh_{i,j}$. With this instantiation, lemma 4.2.5.1 gives us what we need to discharge the assumptions, and we obtain the result $x\sharp\widetilde{U_{i,j}}$.

To infer $x\sharp\widetilde{T}$ and complete the proof, we need another lemma:

Lemma 4.2.5.2 $\widetilde{T} = \widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}}$.

Proof For each $\widetilde{y_{i,j}}$, we have $\widetilde{y_{i,j}} = supp(e_{i,j}) \cap \widetilde{x}$ by lemma 4.2.5.1. Thus, $\widetilde{y_{i,1}} \cup ... \cup \widetilde{y_{i,c_i}} = (supp(e_{i,1}) \cap \widetilde{x}) \cup ... \cup (supp(e_{i,c_i}) \cap \widetilde{x}) = (supp(e_{i,1}) \cup ... \cup supp(e_{i,c_i})) \cap \widetilde{x} = supp(C_i\ e_{i,1}\ ...\ e_{i,c_i}) \cap \widetilde{x}$. Since $\widetilde{x} \subseteq supp(C_i\ e_{i,1}\ ...\ e_{i,c_i})$, it follows that $\widetilde{y_{i,1}} \cup ... \cup \widetilde{y_{i,c_i}} = \widetilde{x}$.

For each $U_{i,j}$, lemma 4.2.5.1 gives us $U_{i,j} = \{\widetilde{T_n}.\widetilde{x}_n \in \widetilde{y_{i,j}}\}$. Thus, $\widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}} = \{\widetilde{T_n}.\widetilde{x}_n \in \widetilde{y_{i,1}}\} \cup ... \cup \{\widetilde{T_n}.\widetilde{x}_n \in \widetilde{y_{i,c_i}}\} = \{\widetilde{T_n}.\widetilde{x}_n \in (\widetilde{y_{i,1}} \cup ... \cup \widetilde{y_{i,c_i}})\} = \{\widetilde{T_n}.\widetilde{x}_n \in \widetilde{x}\} = \widetilde{T}$. ∎

With lemma 4.2.5.2 in place, we can complete the proof as follows:

$$\begin{aligned} x\sharp\widetilde{T} &= x\sharp(\widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}}) \\ &= (x\sharp\widetilde{U_{i,1}}) \wedge ... \wedge (x\sharp\widetilde{U_{i,c_i}}) \\ &= True \end{aligned}$$

The implementation as a tactic consists of several different subroutines, each employing its own tactics to handle a particular part of the proof. We will describe them in a bottom-up fashion:

Obtain procedure

Given a component $e_{i,j}$ of Y, and the corresponding instance of lemma 4.2.5.1, this procedure obtains fixed sequences $\widetilde{y_{i,j}}$ and $\widetilde{U_{i,j}}$ with all the properties that follow from lemma 4.2.5.1, and returns a new proof context in which those sequences have been fixed, and their properties registered as assumptions.

Induction procedure

Given a component $e_{i,j}$ of Y that is a substitution type, and the results of calling the obtain procedure for the same $e_{i,j}$, applies the corresponding freshness theorem (or induction hypothesis) $fresh_{i,j}$ to obtain the result $x\sharp\widetilde{U_{i,j}}$. The tactic applies $fresh_{i,j}$ to the goal $x\sharp\widetilde{U_{i,j}}$, and uses the results from the obtain procedure to discharge the new subgoals.

Given a component $e_{i,j}$ of Y that does not carry names, the result $x\sharp\widetilde{U_{i,j}}$ is also obtained, but since there is no corresponding $fresh_{i,j}$ property, it is inferred from the fact that $supp(e_{i,j}) = \{\}$. The tactic for handling this inserts the results of the obtain tactic as assumptions, followed by a call to the simplifier.

In both cases, the theorem $x\sharp\widetilde{U_{i,j}}$ is returned.

Union procedure

From the results of applying the obtain and induction procedures to each component $e_{i,j}$ of a constructor C_i, this procedure proves lemma 4.2.5.2. This is handled by the simplifier, which performs two steps:

1. Prove the lemma $\widetilde{x} \subseteq (\widetilde{y_{i,1}} \cup ... \cup \widetilde{y_{i,c_i}})$ from the theorems $\widetilde{y_{i,1}} = \mathit{supp}(e_{i,1}) \cap \widetilde{x_{i,1}}$, ..., $\widetilde{y_{i,c_i}} = \mathit{supp}(e_{i,c_i}) \cap \widetilde{x_{i,c_i}}$ and $\widetilde{x} \subseteq \mathit{supp}(C_i\, e_{i,1} \ldots e_{i,c_i})$. This simpset uses the above theorems as well as some basic properties of $\cap$ and $\cup$.

2. Prove $\widetilde{T} = \widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}}$, using this lemma and the results $\widetilde{U_{i,1}} = \{\widetilde{T}_n.\widetilde{x}_n \in \widetilde{y_{i,1}}\}$, ..., $\widetilde{U_{i,c_i}} = \{\widetilde{T}_n.\widetilde{x}_n \in \widetilde{y_{i,c_i}}\}$. The simpset also includes the key theorem $\{\widetilde{T}_n.\widetilde{x}_n \in A\} \cup \{\widetilde{T}_n.\widetilde{x}_n \in B\} = \{\widetilde{T}_n.\widetilde{x}_n \in (A \cup B)\}$.

The returned result is the theorem $\widetilde{T} = \widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}}$.

Wrapup procedure

With the result $\widetilde{T} = \widetilde{U_{i,1}} \cup ... \cup \widetilde{U_{i,c_i}}$ from the union procedure, and the results $x \sharp \widetilde{U_{i,1}}$, ..., $x \sharp \widetilde{U_{i,c_i}}$ from the induction procedure, this procedure obtains the result $x \sharp \widetilde{T}$. The tactic inserts the results of the union and induction procedures as assumptions, followed by a call to the simplifier. The simpset used contains the key theorem $a \sharp (X \cup Y) = (a \sharp X) \wedge (a \sharp Y)$[1].

A theorem stating that $x \sharp \widetilde{T}$ is returned.

Single tactic

This procedure ties together the obtain, induction, union and wrapup procedures to perform the complete proof for a single constructor C_i of Y. The obtain and induction procedures are applied once for each component $e_{i,j}$ of C_i, and the results obtained go through the union and wrapup procedures to obtain the theorem $x \sharp \widetilde{T}$, which is the desired result.

However, one more step remains: the obtain tactic introduces a new proof context with additional assumptions, that the theorem $x \sharp \widetilde{T}$ depends on. Thus, this theorem cannot be applied directly in the original context,

[1]This theorem only holds if X and Y are finite - however, this is not an issue here since we only use finite sequences.

which does not contain these assumptions. First, the theorem $x \sharp \widetilde{T}$ must be exported to the original context.

This tactic then applies the exported $x \sharp \widetilde{T}$ result as a backwards rule, completing the proof.

Freshness tactic

Finally, the freshness tactic performs the whole proof - it applies induction over y to the goal state, generalising over $\widetilde{x}$ and $\widetilde{T}$. The single tactic is then applied to each new subgoal.

4.2.6 Other proofs

As previously stated, the main focus of this work is to automate the proofs of the substitution lemmas, since those proofs concern low-level details that a user ideally shouldn't have to bother with, and seem to share a common structure that is fairly independent of the particular instance being studied. The same cannot be said about the other requisites on a psi-instance.

Nonetheless, tactics are implemented for these proofs - the idea being that at the very least, any proof obligaition which is amenable to a simple proof such as

apply(*induct x*)
by(*simp+*)

should be handled in a behind-the-scenes manner. To this end, all proof obligations $P(v_1, ..., v_i)$ except the substitution lemmas, where $v_1, ..., v_i$ are the free variables of P, are subjected to a proof attempt with a tactic that does the following:

1. Unfold the definition of static equivalence.
2. Unfold the definition of static implication.
3. For each $v \in \{v_1, ..., v_i\}$, apply induction over v to all subgoals.
4. Apply the simplifier to all subgoals.

Here we employ the user-level simpset, which contains the default HOL simpset as well as any simplification rules declared by the user[2]. When proving equivariance properties, equivariance theorems are also included.

4.3 Example instances

4.3.1 The pi-calculus

The pi-calculus [17] is the original mobile process calculus, featuring names in the dual role of communication channels and data terms. Its formulation as a psi-instance is given in [15, page 46] as:

$$\begin{array}{lcl}
\mathbf{T} & \triangleq & \mathcal{N} \\
\mathbf{C} & \triangleq & \{a = b . a, b \in \mathbf{T}\} \cup \{\mathrm{T}\} \\
\mathbf{A} & \triangleq & \{e\} \\
\dot{\leftrightarrow} & \triangleq & = \\
\otimes & \triangleq & \lambda \Psi_1, \Psi_2 . e \\
\mathbf{1} & \triangleq & e \\
\vdash & \triangleq & \{(\mathbf{1}, a = a) . a \in \mathcal{N}\} \cup \{\mathbf{1}, T\}
\end{array}$$

Encoding T as $a = a$ for any choice of a, the datatypes of this instance can be encoded in Isabelle as follows:

nominal_datatype $term = T\, name$
nominal_datatype $cond = Eq\, term\, term$
nominal_datatype $assert = A\, unit$

The definition of the substitution functions is straightforward:

[2]For an example, with the [*simp*] attribute when stating theorems

```
interpretation nameSubst "t :: term" done

nominal_primrec termSubst
:: "term ⇒ name list ⇒ term list ⇒ term"
where
  "termSubst (T a) xvec Tvec =
     cases substName a xvec Tvec of names ⇒ T | terms ⇒ id"


nominal_primrec condSubst
:: "cond ⇒ name list ⇒ term list ⇒ cond"
where
  "condSubst (Eq M N) xvec Tvec =
     Eq (M[xvec := Tvec]) (N[xvec := Tvec])"


nominal_primrec assertSubst
:: "assert ⇒ name list ⇒ term list ⇒ assert"
where
  "assertSubst (A ()) xvec Tvec = A ()"
```

The definition of entailment is simple since there's only one assertion, but looks rather convoluted. This is because nominal datatypes are not equality types - hence, pattern matching and equality comparison is not always possible. Here, the comparison must be distributed onto the base case of names:

```
fun entail :: "assert ⇒ cond ⇒ bool"
where
  "entail _ c =
     (∀a b.((c = Eq (T a) (T b) → a = b)"
```

The encoding of the three remaining morphisms is trivial and will be omitted. Given these definitions, the pi-calculus is automatically verified by a single pass of the **psi_instance** command, provided the user supplies a manual proof of the *Subst3Aux* lemma for terms (see section 4.2.5).

4.3.2 The polyadic pi-calculus

The polyadic pi-calculus [16] extends the pi-calculus with tupling symbols $t_n(a_1, ..., a_n)$, where $a_1, ..., a_n \in \mathcal{N}$. While such tuples cannot act as communication channels, they allow multiple names to be transmitted with a single action. Its encoding as a psi-instance [15, pp. 63-64] is obtained by modifying the pi-calculus instance from section 4.3.1 as follows:

$$\mathbf{T} \triangleq \mathcal{N} \cup \{t_n(a_1, ..., a_n), a_1, ..., a_n \in \mathcal{N}\}$$

This can be encoded in Isabelle as sequences of terms:

nominal_datatype *term* = *T name* | *S name term*

When defining the term substitution function, there is a minor issue that must be circumvented. Since the psi-calculus framework requires substitution functions that substitute names for arbitrary terms, we must have a means of substituting names for tupling symbols, as in:

$t_n(x_1, ..., x_n)[\langle x_i \rangle := \langle t_m(y_1, ..., y_m) \rangle]$, where $1 \leq i \leq n$

This is strictly a technicality, since no such substitution can ever occur during a transition, yet simply ignoring the substitution is not an option since that behaviour would violate the substitution lemmas. One solution is to let $t_n(x_1, ..., x_n)[\langle x_i \rangle := \langle t_m(y_1, ..., y_m) \rangle] = t_n(x_1, ..., x_{i-1}, y_1, ..., y_m, x_{i+1}, ..., x_n)$. To encode this, we define concatenation over terms:

nominal_primrec *termConcat* :: "*term* ⇒ *term* ⇒ *term*"
where
  "*termConcat* (*T a*) *N* = *S a N*" |
  "*termConcat* (*S a M*) *N* = *S a* (*termConcat M N*)"

Term substitution is then defined as follows:

nominal_primrec $termSubst$
$::$ "$term \Rightarrow name\,list \Rightarrow term\,list \Rightarrow term$"
where
  "$termSubst\ (T\ a)\ xvec\,Tvec =$
    $cases\ substName\,a\,xvec\,Tvec\,of\ names \Rightarrow T \mid terms \Rightarrow id$"
  "$termSubst\ (S\ a\ M)\ xvec\,Tvec =$
    $termConcat$
      $(cases\ substName\,a\,xvec\,Tvec\,of\ names \Rightarrow T \mid terms \Rightarrow id)$
      $(termSubstMxvecTvec)$"

The proof heuristics for the substitution lemmas detailed in sections 4.2.3, 4.2.4 and 4.2.5 assume that substitution over a datatype simply pushes the substitution onto the datatype's components, which isn't strictly the case here: in addition, we also apply a concatenation function, which the proof heuristics doesn't have any information about. Hence, it will be unable to verify the term substitution lemmas without additional information.

To be able to verify this instance automatically, the user must first manually prove that *termConcat* distributes permutation, freshness and support onto its components:

lemma $termConcatSupp:$
  "$supp(termConcat\ M\ N) = (supp(M) \cup (supp(N) :: name\ set))$"
by$(nominal\_induct\ M\ rule:\ term.strong\_inducts,$
    $(force\ simp\ add:\ term.supp)+)$

lemma $termConcatEqvt[eqvt]:$
  "$p \cdot (termConcat\ M\ N) = termConcat(p \cdot M)(p \cdot N)$"
by$(nominal\_induct\ M\ rule:\ term.strong\_inducts, simp+)$

lemma $termConcatFresh:$
  "$x\sharp(termConcat\ M\ N) = x\sharp M \wedge x\sharp N$"
by$(nominal\_induct\ M\ rule:\ term.strong\_inducts,$
    $(simp\ add:\ term.fresh)+)$

These must then be declared as simplification rules about terms when applying the **psi_instance** command.

If the user supplies a manual proof of *Subst3Aux* for terms (see section

4.2.5), the tool can now discharge all proof obligations automatically, with the exception of channel symmetry and channel transitivity.

4.3.3 The pi-F calculus

The pi-F calculus is introduced by Wischik and Gardner in [29]. It features explicit fusions of the form $a = b$ as agents, and fuses the names in object position during communication. An encoding of pi-F as a psi-instance is given in [9] as:

$$\begin{array}{lcl}
\mathbf{T} & \triangleq & \mathcal{N} \\
\mathbf{C} & \triangleq & \{a = b.a, b \in \mathbf{T}\} \cup \{a \dot{\leftrightarrow} b.a, b \in \mathbf{T}\} \\
\mathbf{A} & \triangleq & \{\{a_1 = b_1, ..., a_n = b_n\}.a_i \in \mathcal{N}, b_i \in \mathcal{N}\} \\
\dot{\leftrightarrow} & \triangleq & \lambda x, y.x \dot{\leftrightarrow} y \\
\otimes & \triangleq & \cup \\
\mathbf{1} & \triangleq & \{\} \\
\vdash & \triangleq & \{(\Psi, a = b).a = b \in EQ(\Psi)\} \cup \{(\Psi, a \dot{\leftrightarrow} b).\Psi \vdash a = b\}
\end{array}$$

where $EQ(\Psi)$ is the equivalence closure of Ψ. We first observe that having conditions specifically for representing channel equivalence is unnecessary. Since $\Psi \vdash a = b$ iff $\Psi \vdash a \dot{\leftrightarrow} b$, we can make do with only $=$. We thus simplify $\mathbf{C}$ and $\vdash$ as follows:

$$\begin{array}{lcl}
\mathbf{C} & \triangleq & \{a = b.a, b \in \mathbf{T}\}\} \\
\vdash & \triangleq & \{(\Psi, a = b).a = b \in EQ(\Psi)\} \\
\dot{\leftrightarrow} & \triangleq & =
\end{array}$$

Encoding the finites sets of fusions in **A** as lists of fusions, the encoding of the nominal datatypes and their substitution functions is identical to the encoding of the pi-calculus in section 4.3.1, with the exception of assertions:

nominal_datatype $assert = E\ unit \mid A\ term\ term\ assert$

nominal_primrec *assertSubst*
:: "*assert* ⇒ *name list* ⇒ *term list* ⇒ *assert*"
where
  "*assertSubst* $(E\ ())$ *xvec Tvec* $= E\ ()$" |
  "*assertSubst* $(A\ M\ N\ \Psi)$ *xvec Tvec* $=$
    $A\ (M[xvec := Tvec])\ (N[xvec := Tvec])\ (A[xvec := Tvec])$"

The encodings of $\dot{\leftrightarrow}$ and **1** are straightforward. In order to encode the composition operator $\otimes$, we require a definition of concatenation, analogous to the term concatenation defined for the polyadic pi-calculus in section 4.3.2:

nominal_primrec *assertConcat* :: "*assert* ⇒ *assert* ⇒ *assert*"
where
  "*assertConcat* $(E\ a)\ \Psi = \Psi$" |
  "*assertConcat* $(A\ M\ N\ \Psi')\ \Psi = A\ M\ N$ (*assertConcat* $\Psi'\ \Psi$)"

fun *composition* :: "*assert* ⇒ *assert* ⇒ *assert*"
where
  "*composition* $\Psi\ \Psi'$ = *assertConcat* $\Psi\ \Psi'$"

For the entailment relation $\vdash$, we need a notion of equivalence closure, which is not supplied by the Isabelle library. Moreover, we require our equivalence closure function to be equivariant. To this end, we define an inductive predicate corresponding to membership in the equivalence closure of a set $S \subseteq \mathcal{N} \times \mathcal{N}$. Equivariance lemmas and other infrastructure for nominal reasoning about this relation is then provided automatically by the **equivariance** and **nominal_inductive** commands of the nominal datatype package:

```
inductive $eq\_name\_close$
:: "$(name \times name) \Rightarrow (name \times name)\ set \Rightarrow bool$"
where
  "$(a, b) \in S \Rightarrow eq\_name\_close\ (a, b)\ S$" |
  "$eq\_name\_close\ (a, a)\ S$" |
  "$eq\_name\_close\ (a, b)\ S \Rightarrow eq\_name\_close\ (b, a)\ S$" |
  "$[\![eq\_name\_close\ (a, b)\ S;\ eq\_name\_close\ (b, c)\ S]\!]$
    $\Rightarrow eq\_name\_close\ (a, c)\ S$"

equivariance $eq\_name\_close$

nominal_inductive $eq\_name\_close$ done
```

Entailment can then be defined as follows:

```
fun $entail$ :: "$assert \Rightarrow cond \Rightarrow bool$"
where
  "$entail\ \Psi\ \varphi =$
    $eq\_name\_close\ (pair\_of\ \varphi)\ (set(pairs\_of\,\Psi))$"
```

where $pair\_of$ and $pairs\_of$ implement isomorphisms from **C** to $\mathcal{N} \times \mathcal{N}$, and from **A** to $seq(\mathcal{N} \times \mathcal{N})$, respectively.

Given these definitions, along with manual proofs of the *Subst3Aux* property for terms and assertions (see section 4.2.5), a single pass of the **psi_instance** command will prove all the substitution lemmas and all equivariance properties. The six remaining proof obligations must be proven manually.

Chapter 5

Conclusion

5.1 Method

5.1.1 Learning Isabelle

A major hurdle to overcome in the early days of the project was to learn how to work with Isabelle, both on the user level and the ML-level.

Given the reputation of theorem provers as hard to use, I found the learning curve of Isabelle's user level was not anywhere near as steep as I expected. The rich flora of documentation and tutorials available, an active mailing list, an experienced supervisor and some prior exposure to formal methods all helped make the learning process mostly smooth going. The method of undertaking a smaller practice project was very helpful, and gave me a solid foundation of Isabelle knowledge for the main project.

Learning Isabelle programming, however, turned out to be a much more difficult endeavour. Being already fluent in Standard ML, I had a bit of a head start. But Isabelle is a large system, showing many signs of having evolved organically over many years. As a result, its structure looks quite intimidating to the newcomer, with every module depending on yet more esoteric modules, making it difficult to know where to even begin. While some documentation exists, it is not anywhere near as complete as the documentation for the user level. In particular, internal documentation of the source code is very sparse.

The approach taken to learning Isabelle programming was to just jump straight into the project without any real prior knowledge, using the examples provided in the Isabelle Cookbook [11] as initial building blocks, figuring things out as I went along. Thus, the overlap between the learning process and the development process was significant. While this was slow going in the beginning, it was certainly viable. In retrospect, I would have preferred to undertake a smaller practice project before beginning development, as when learning to work with the Isabelle user level. While such an approach could have been more expensive in terms of time, it would have provided a harmless context in which to make the inevitable beginner's mistakes. Some unnecessarily convoluted design decisions and problem solutions that I came up with could have been avoided. Additionally, throughout the project I often had the uneasy feeling of understanding the how's but not the why's of Isabelle programming, something that a bit of prior practical experience would almost certainly have helped with.

5.1.2 Development

The development process was mostly characterized by improvisation and organic growth, rather than careful planning. This method certainly worked for the purposes of this project, but I feel that imposing more structure on the development process would have been effective.

For a project like this, it seems inevitable that a significant amount of free-form exploration must be part of it. While this is of course not a bad thing in and of itself when developing research software, it can be expensive in terms of time as opposed to a more structured process, and the code easily becomes chaotic and difficult to maintain.

A concrete example where more structure would have helped is the division of labour between me and Palle Raabjerg. We often worked on different versions of the same file in parallel without much coordination. When he had finished a piece of code, it would be copied from his version into mine. A not insignificant amount of effort would then have to be spent adapting his code to make it fit into the overall framework, effort that would be unnecessary with a more structured development process.

Occasional bursts of structured development did occur during the project, with dependencies, calling hierarchies and function signatures sketched in writing before any actual code was written. These were by far the most effective periods of the development process, which seems to support my feeling that the balance between structured development and free-form exploration would have worked better if weighted more towards structure.

5.1.3 Project structure

As discussed in section 2.2, the original plan was to develop tools and example instances in parallel. The idea was to start from a trivial instance and develop the tool to the point where this instance could be verified automatically, and from that point, proceed to gradually more interesting example instances, making the tool more sophisticated in order to accomodate their verification.

This turned out to be infeasible, mostly because of the unexpected complexity of Isabelle programming, and the sheer size of the infrastructure needed to verify any instance. Instead, most of the project was devoted to developing the tool, with the study of example instance being delegated to the tail-end of the project.

A more fruitful approach might have been to reverse this ordering - beginning with example instances, and then developing the tool. When developing proof heuristics such as those described in section 4.2, certain assumptions about the instances that don't hold in the general case must be made. For an example, the substitution functions must be structured in a certain way for the proof methods to apply. Ideally, these assumptions should be properties that many applications of psi-calculi have in common. Thus, having more practical examples of psi-instances and their Isabelle encodings at hand at the start of development, would certainly have provided a more solid understanding of what assumptions about instances are suitable in practice.

At the start of the tool's development, the only instance that was encoded in Isabelle was the pi-calculus, and the assumptions made about instances are properties that hold for the pi-calculus and instances with a similar structure. Some of these are probably unnecessarily strong - for an example, the assumption that the substitution functions operate by distributing the substitution onto the datatype components.

5.2 Results

We have developed a tool which significantly reduces the amount of manual work necessary to formally verify psi-instances. For an example, a theory file by Bengtson developed before the start of this project, which encodes the pi-calculus as a psi-instance and verifies it, consists of 625 lines of Isabelle code. Using the tool, performing the same task requires only 74 lines of code.

Of course, the tool has plenty of room for extensions and improvements.

For an example, one of the design goals of the tool was to make the verification of the substitution lemmas fully automatic. For instances on the form described in section 4.2.1, this has been achieved, with one exception: the *Subst3Aux* lemma described in section 4.2.5, which is not currently automated and must be supplied manually by the user.

From a usability perspective, it is desirable for the proof obligations presented to the user to be natural, in two senses - they should be easy to understand, and it should be easy for the user to see how they connect to the big picture. *Subst3Aux* isn't very natural in either of these senses. However, with the exception of *Subst3Aux*, all proof obligations presented to the user correspond exactly to the requisites on a psi-instance, as detailed in section 1.4.

As seen with the example of the polyadic pi-calculus in section 4.3.2, the assumption that the substitution function pushes the substitution onto the datatype components is not strictly necessary for the proof strategies to work. It suffices for this to hold up to support - for an example, $supp((T\,n\,m)[\widetilde{x} := \widetilde{M}]) = supp(n[\widetilde{x} := \widetilde{M}]) \cup supp(m[\widetilde{x} := \widetilde{M}])$. In the polyadic pi-calculus example, the user must manually verify these properties. Detecting the need for such lemmas and developing proof heuristics for them should be a straight-forward extension of the tool.

Another desirable extension would be to have the tool define substitution functions automatically if the user does not provide them. Also, the tool could be generalised to incorporate support for library datatypes such as lists and options, as well as mutually recursive datatypes.

Lack of time is the main reason why most of these features are not in place. In the case of the *Subst3Aux* lemma, no proof strategy which generalises well to arbitrary instances has been found.

The source code of the tool has been written with an eye towards easily accomodating future extensions - thus, most of the extensions described above would be straight-forward to incorporate into the existing infrastructure, and the infrastructure should be resilient to future extensions or revisions of the psi-calculus framework. The one exception is support for mutually recursive datatypes. The absence of mutually recursive datatypes is assumed not only in the proof methods, but in large parts of the infrastructure. Incorporating support for it would require extensive revisions of the code.

The three example instances provided does serve the purpose of illustrating the expressiveness of the tool. However, they are all based on instances that were already discussed in [9] and [15], though not verified formally. Thus, these examples do not provide any novel insights on the expressiveness of the psi-calculus framework itself. One reason for this is lack of time, as a consequence of the unexpectedly large amount of groundwork needed to get the tool going. Another factor is the lack of any prior knowledge on my part about process calculi and their applications upon starting this project. More prior experience with process calculi would probably be necessary before defining interesting, novel instances.

5.3 Related works

Previous work on tool support for process calculi has mostly focused on protocol verification and behavioural equivalence checking. The Concurrency Workbench [18] by Moller and Stevens is a tool that facilitates various equivalence, preorder and model checking in CCS or SCCS. The Mobility Workbench, developed by Victor, Moller, Dam and Eriksson, offers similar functionality for both the monadic [27] and polyadic [25] pi-calculus, as well as hyper-equivalence checking for the fusion calculus [26]. ProVerif [12], an automatic verifier for cryptographic protocols developed by Blanchet, can verify protocols described as processes in an extension of the pi-calculus [1].

Some previous work on tool support for process calculi conducted within theorem provers is also available. In [4], Aït Mohamed presents an automatic HOL tactic for checking early equivalence between agents in a pi-calculus without recursion. The tactic can be applied to agents with recursion, but is not guaranteed to terminate in that case. It works by alternating between rewriting the terms into prefixed form, and applying rules based on the definition of the bisimulation relation to the prefixed terms.

In [13], Briais proposes a Coq formalisation of the spi-calculus that will eventually be the basis of a certified tool for automatic equivalence checking. Additionally, a tool for automatically translating protocol narrations into spi-calculus terms is proposed. It should be noted that Briais's Coq formalisation of the spi-calculus has some interesting similarities to Bengtson's Isabelle formalisation of psi-calculi. They both work with parametrised calculi. Briais defines a generalised pi-calculus, parametrised over types representing the sets of guards and expressions, respectively. These are then instantiated to obtain both the monadic pi-calculus and the spi-calculus. Briais uses de Bruijn indices [14] rather than nominal logic to treat α-equivalence, yet there are about 60 technical results roughly analogous to our substitution lemmas which must be proven for each datatype. How these proofs work is not detailed, however, and no special tools for their automation seem to have been developed.

One difference between the tools described above and this work that makes direct comparison difficult, is that they all focus on verifying properties about processes, as described in their respective calculi. In contrast, this work focuses on verifying properties of calculi.

5.4 Acknowledgements

I am grateful to my supervisor Jesper Bengtson for all his help, and for providing valuable feedback on drafts of this report. Stefan Berghofer has also provided much valuable help in the process of learning Isabelle programming, always having a simpler and more direct approach up his sleeve than whatever convoluted scheme to achieve our goals that we would hatch on our own.

Of course, thanks are also due to Palle Raabjerg, my fellow traveller through much of this journey.

Bibliography

[1] Martín Abadi and Bruno Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, January 2005.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, January 2001.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.

[4] Otmane Aït Mohamed. Mechanizing a π-Calculus Equivalence in HOL. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *LNCS*, pages 1–16, 1995.

[5] Clemens Ballarin. Tutorial to Locales and Locale Interpretation. `http://isabelle.in.tum.de/dist/Isabelle/doc/locales.pdf`.

[6] Clemens Ballarin. Locales and locale interpretation in Isabelle/Isar. In *Types for Proofs and Programs (TYPES 2003)*, volume 3085, pages 34–50. LCNS, Springer, 2004.

[7] H.P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1984.

[8] Jesper Bengtson. *Formalising Process Calculi.* PhD thesis, Uppsala University, 2010.

[9] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: Mobile processes, nominal data, and logic. In *Proceedings of LICS 2009*, pages 39–48. IEEE, 2009.

[10] Jesper Bengtson and Joachim Parrow. Psi-calculi in Isabelle. In *Proceedings of TPHOLs 2009*, volume 4423 of *LNCS*, pages 99–114. IEEE, 2009.

[11] Stefan Berghofer, Jasmin Blanchette, Sascha Böhme, Lukas Bulwahn, Jeremy Dawson, Alexander Krauss, Tobias Nipkow, Andreas Schropp, and Christian Sternagel. The Isabelle Cookbook — A Tutorial for Programming on the ML-level of Isabelle (draft). `http://isabelle.in.tum.de/nominal/activities/idp/`, 2010.

[12] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[13] Sébastien Briais. *Theore and tool suppport for the formal verification of cryptographic protocols*. PhD thesis, École Polytechnique Fédérale de Lausanne, January 2008.

[14] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[15] Magnus Johansson. *Psi-calculi: a Framework for Mobile Process Calculi*. PhD thesis, Uppsala University, 2010.

[16] Robin Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *F*, 1993.

[17] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[18] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). `http://homepages.inf.ed.ac.uk/perdita/cwb/`.

[19] Tobias Nipkow. A Tutorial Introduction to Structured Isar Proofs. `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf`.

[20] Tobias Nipkow. Theorem Proving with Isabelle/HOL — An Intensive Course. `http://isabelle.in.tum.de/coursematerial/PSV2009-1/index.html`, 2009.

[21] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] A.M. Pitts. Nominal logic, a first order theory of names and binding. *Journal of Information and Computation*, 186:165–193, 2003.

[23] Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, May 2008.

[24] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 35–50, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Björn Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[26] Björn Victor. Symbolic characterizations and algorithms for hyperequivalence. Technical Report DoCS 98/96, Department of Computer Systems, Uppsala University, Sweden, December 1998.

[27] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In David Hill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[28] Makarius Wenzel. The Isabelle/Isar Implementation. `http://isabelle.in.tum.de/dist/Isabelle/doc/implementation.pdf`, 2010.

[29] Lucian Wischik and Philippa Gardner. Explicit fusions. *Theoretical Computer Science*, 304(3):606–630, 2005.