

A Framework for the Formalisation of Pi Calculus Type Systems in Isabelle/HOL

Simon J. Gay*

Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK.
Email: <simon@dcs.gla.ac.uk>

Abstract. We present a formalisation, in the theorem proving system Isabelle/HOL, of a linear type system for the pi calculus, including a proof of runtime safety of typed processes. The use of a uniform encoding of pi calculus syntax in a meta language, the development of a general theory of type environments, and the structured formalisation of the main proofs, facilitate the adaptation of the Isabelle theories and proof scripts to variations on the language and other type systems.

Keywords: Types; pi calculus; automatic theorem proving; semantics.

1 Introduction

Static type systems are an established feature of programming languages, and the range of type systems is growing. The general principle is that a program which has been typechecked (at compile time) is guaranteed to satisfy a particular runtime property: for example (traditionally) that all functions are called with the correct number and type of arguments, or (more exotically) that some security property, such as secrecy, holds. In order for typechecking to give a proper guarantee of some form of program correctness, it is essential to prove, with respect to a well-defined semantics, that execution of typed programs is well-behaved.

Proofs of type soundness have traditionally been carried out by hand, which is feasible for small theoretical languages but much more difficult for real programming languages such as Java. The proofs are characterised by large inductions over the syntax of the language, the definition of the typing rules, and the definition of the operational semantics; many cases must be considered, which often differ in subtle ways and are very sensitive to the precise formulation of the syntax and the typing rules. Modifications of the language require proofs to be redone, often with only minor variations but it is dangerous to assume that previously-established properties will survive.

For these reasons, the idea of formalising proofs of type soundness (and other aspects of programming language theory, such as program equivalence, but this is not the concern of the present paper) within an automatic theorem proving system is very attractive. As part of a research project investigating type systems

* Partially supported by EPSRC grants GR/L75177 and GR/N39494.

for the control of concurrent communication, we are interested in concurrent programming languages based on the pi calculus. The pi calculus provides a core concurrent programming notation, dealing with inter-process communication along named channels. We have developed a formalisation within Isabelle/HOL of the pi calculus and the linear type system proposed by Kobayashi *et al.* [12]. Our aim has been not simply to formalise a particular type system, but to develop a flexible framework within which a variety of type systems can be investigated, with as much re-use of existing theory and proof as possible. Our formalisation of the linear type system is a step towards a treatment of a language [5] based on the pi calculus and combining session types [10, 24] with subtyping [20].

We should emphasise that the purpose of this formalisation is to prove properties of the type system itself, principally type soundness, rather than to support typechecking. However, given the formalisation of a pi calculus type system it should be straightforward to construct proofs that particular processes are typable.

Related Work

The following table summarises work on formalisation of the pi calculus by a number of researchers, including the present paper. The parameters are: the theorem proving system used; the style of pi calculus (monadic or polyadic); the approach to binding (names, de Bruijn indices [1], higher order abstract syntax (HOAS) [19], McKinna and Pollack’s [13] approach (MP), Gabbay and Pitts’ [4] FM-sets approach (GP)); the style of operational semantics (labelled transition system (LTS) or reduction relation (RED)); the focus on bisimulation (\sim) or typing (\vdash).

Author	Prover	Calculus	Binding	Semantics	Focus
Melham [14]	HOL	monadic	names	LTS	\sim
Hirschhoff [9]	Coq	polyadic	de Bruijn	LTS	\sim
Honsell <i>et al.</i> [11]	Coq	monadic	HOAS	LTS	\sim
Röckl <i>et al.</i> [22]	Isabelle/HOL	monadic	HOAS	-	syntax
Röckl [21]	Isabelle/HOL	monadic	GP	-	syntax
Henry-Gréard [8]	Coq	monadic	MP	LTS	\vdash
Despeyroux [2]	Coq	monadic	HOAS	LTS	\vdash
This paper	Isabelle/HOL	polyadic	de Bruijn	RED	\vdash

There are two main novelties of the formalisation reported in the present paper. First, we formalise an operational semantics based on a reduction relation rather than a labelled transition system; this is because the language which we set out to formalise uses the reduction semantics. Second, we use a meta language rather than a direct formalisation of pi calculus syntax. More technically, the introduction of a type system for the meta language (see Section 8) is also new. The meta language approach supports our goal of developing a general framework in which a variety of languages and type systems can be formalised; we return to this point in Section 12.

Formalisations of other programming language type systems have been carried out, notably for Java [26, 23] and Standard ML [3]. The present paper

concentrates on concurrent languages in the style of the pi calculus; the ambitious goal of applying our meta language approach to a wider range of language paradigms is left for future work.

Acknowledgements

The author is grateful to Tom Melham and the anonymous referees for many valuable comments, to Malcolm Hole for discussions of pi calculus type soundness proofs, and to Simon Ambler for discussions of formalisations of operational semantics.

2 Pi Calculus

We assume some familiarity with the pi calculus [16, 17] but summarise its syntax and semantics in this section.

The following grammar defines *processes* P ; variables x stand for channel names; \tilde{x} is a list of variables. There are two binding operators: the variables \tilde{x} are bound in $(\nu\tilde{x})P$ and in $x?[\tilde{x}].P$. All other variable occurrences are free.

$$\begin{array}{l|l|l}
 P ::= \mathbf{0} & (\text{nil}) & | (\nu\tilde{x})P \quad (\text{restriction}) \\
 | P | P & (\text{parallel}) & | x?[\tilde{x}].P \quad (\text{input}) \\
 | !P & (\text{replication}) & | x![\tilde{x}].P \quad (\text{output}) \\
 | P + P & (\text{choice}) &
 \end{array}$$

The operational semantics of the pi calculus is defined in two stages. The *structural congruence* relation axiomatises some basic process equivalences, and the *reduction relation* axiomatises inter-process communication.

The structural congruence relation, denoted by \equiv , is defined inductively as the smallest equivalence relation which is additionally a congruence (i.e. is preserved by the process constructors) and is closed under a number of rules. These rules include α -equivalence, basic properties of parallel composition and choice (commutativity, associativity, the fact that $\mathbf{0}$ is a unit), and some less trivial equivalences. Chief among these are a rule which captures the meaning of replication:

$$!P \equiv P!P$$

and the *scope extrusion* rule:

$$P | (\nu\tilde{x})Q \equiv (\nu\tilde{x})(P | Q) \quad \text{if none of the } \tilde{x} \text{ are free in } P.$$

The reduction relation is also defined inductively, by the following rules. The first rule uses substitution to express transmission of channel names during communication; only substitution of variables for variables is required.

$$\begin{array}{c}
 \frac{\text{length}(\tilde{y}) = \text{length}(\tilde{z})}{x?[\tilde{y}].P | x![\tilde{z}].Q \longrightarrow P\{\tilde{z}/\tilde{y}\} | Q} \qquad \frac{P \longrightarrow P'}{(\nu\tilde{x})P \longrightarrow (\nu\tilde{x})P'} \\
 \\
 \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'} \qquad \frac{P \longrightarrow P'}{P | Q \longrightarrow P' | Q} \qquad \frac{P \longrightarrow P'}{P + Q \longrightarrow P'}
 \end{array}$$

Communication between an input and an output of different arities is not defined, and the appearance of such a miscommunication during the execution of a process can be interpreted as a runtime error, analogous to calling a function with the wrong number of arguments. The simple type system [15, 25] for the pi calculus allows such errors to be eliminated by static typechecking; other type systems, such as Kobayashi *et al.*'s linear type system [12] which we formalise in this paper, additionally check other properties of processes. Many pi calculus type systems have the following general features. The syntax of the language is modified so that bound variables are annotated with types, in the Church style. Typing judgements of the form $\Gamma \vdash P$ are used, where Γ is a list $x_1 : T_1, \dots, x_n : T_n$ of typed variables and P is a process, and the set of valid judgements is defined inductively. Note that a process is either correctly typed or not; processes themselves are not assigned types.

Linear Types

Kobayashi *et al.* [12] proposed a linear [6] type system for the pi calculus, in which certain channels must be used for exactly one communication. The main motivation was the observation that many pi calculus programming idioms involve single-use channels, and making this information available to a compiler by means of the type system could permit many useful optimisations. If a particular channel is used for exactly one communication, it must occur once in an input position and once in an output position. It therefore becomes necessary to consider the two ends of a channel separately, and enforce linear use of each end.

Kobayashi *et al.* introduce a system of *multiplicities*: 1 for linear and ω for non-linear, and *polarities*: subsets of $\{i, o\}$ representing input and output capabilities. Types are defined by $T ::= p^m[\tilde{T}]$ and the abbreviations $\uparrow = \{i, o\}$, $? = \{i\}$, $! = \{o\}$, $| = \{\}$ are used for polarities.

If $x : ?^1[\tilde{T}]$ appears in an environment Γ then a process typed in Γ must use the input capability of x , either by receiving a tuple of values (of types \tilde{T}) along x , or by sending x to another process as part of a tuple of values. Similarly, there are two possible ways of using the output capability of a channel. If $x : \uparrow^1[\tilde{T}]$ appears in Γ , then a process typed in Γ must use both capabilities of x .

The typing rule for parallel composition is

$$\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma + \Delta \vdash P \mid Q} \text{T-PAR}$$

where the $+$ operation combines the capabilities of the channels in Γ and Δ ; it is defined on single types (and undefined for unequal multiplicities) by

$$\begin{aligned} p^\omega + q^\omega &= (p \cup q)^\omega \\ p^1 + q^1 &= (p \cup q)^1 \text{ if } p \cap q = \emptyset \end{aligned}$$

and extended to environments with equal domains by

$$(\Gamma_1, x : p^m[\tilde{T}]) + (\Gamma_2, x : q^n[\tilde{T}]) = (\Gamma_1 + \Gamma_2), x : (p^m + q^n)[\tilde{T}].$$

The $+$ operation is also used to add a single capability to a channel in an environment: $(\Gamma, x : p^m[\tilde{T}]) + x : q^n[\tilde{T}] = \Gamma, x : (p^m + q^n)[\tilde{T}]$.

The typing rules for input and output are

$$\frac{\Gamma, \tilde{y} : \tilde{T} \vdash P}{\Gamma + x : !^m[\tilde{T}] \vdash x?[\tilde{y} : \tilde{T}]. P} \text{T-IN} \quad \frac{\Gamma \vdash P}{\Gamma + x : !^m[\tilde{T}] + \tilde{y} : \tilde{T} \vdash x![\tilde{y}]P} \text{T-OUT}$$

Because $+$ is a partial operation, all typing rules implicitly carry the additional hypothesis that all uses of $+$ are defined.

The uniform definition of $+$ on linear and non-linear types means that the same typing rule covers (say) output on both linear and non-linear channels. In the non-linear case, it is possible for Γ to already contain x and \tilde{y} , in which case the hypothesis that $+$ is defined reduces to a check that x and \tilde{y} have the correct types in Γ .

Because $\mathbf{0}$ uses no channels, it is only typable in an environment in which all linear channels have polarity $|$. Such an environment is said to be *unlimited*. Replication can only be used in an unlimited environment, otherwise the possibility would be created of using linear channels more than once.

The typing rule for choice has an *additive* (in the terminology of linear logic [6]) form:

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P + Q} \text{T-SUM}$$

This is because executing $P + Q$ means executing *either* P *or* Q , so the capabilities in Γ will only be used once.

For example, a process which sends the output end of a linear channel along a non-linear channel can be typed as follows, where T is an arbitrary type; note that Γ is unlimited.

$$\frac{\Gamma = x : \downarrow^\omega[!^1[T]], y : !^1[T] \vdash \mathbf{0}}{\Gamma + x : !^\omega[!^1[T]] + y : !^1[T] \vdash x![y]. \mathbf{0}} \text{T-OUT}$$

The final typing judgement is

$$x : \downarrow^\omega[!^1[T]], y : !^1[T] \vdash x![y]. \mathbf{0}.$$

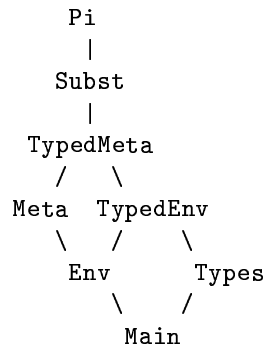
The process $x![y]. \mathbf{0}$ delegates the responsibility for using the output capability of y to whichever process receives it.

We formalise a language slightly different from that of Kobayashi *et al.*: we use output prefix rather than asynchronous output, choice rather than a conditional construct (and therefore drop the boolean type), and replication rather than replicated input. These differences do not change the nature of the type soundness proofs, but simplify some details.

3 Overview of the Formalisation

We have formalised the linear type system for the pi calculus within the theorem prover Isabelle/HOL [18]. Isabelle is a generic theorem proving system; one

of the logics which it implements is HOL, or higher-order logic. Defining a theory in Isabelle/HOL feels very much like programming in a functional language with a rich type system including polymorphism, algebraic datatypes, and extensible record types. Additionally, it is possible to define sets inductively (we make essential use of this feature) or coinductively, and to use (possibly non-computable) logical operators within definitions. Isabelle provides powerful automatic proof tactics, including simplification (generalised equational rewriting), classical reasoning, various kinds of resolution, induction, and case-analysis.



All the definitional mechanisms are associated with appropriate proof rules: making an inductive definition of a set automatically defines elimination rules and an induction rule to support proofs by case-analysis or induction. As well as using the proof tactics defined by Isabelle, it is possible to define (by programming in Standard ML) new tactics; this is made easy by Isabelle's provision of a range of *tacticals*, or combinators for tactics, such as THEN for sequencing and REPEAT for repetition.

The formalisation consists of a hierarchy of Isabelle theories. Each theory has a definition file and a proof file. The hierarchy is as shown; each theory depends on the theories below it. `Main` is predefined and provides all the definitional facilities mentioned above. In the following sections, we describe each theory in turn. Wherever possible, the theories are defined in a sufficiently general way to support the formalisation of different pi calculus type systems.

The theory files and proof scripts are available from the following web address: <http://www.dcs.gla.ac.uk/~simon>. The complete formalisation consists of approximately 650 lines of theory definitions and 6500 lines of proof script.

4 The Theory Types

`Types` is a generic theory of linear and non-linear types, which accounts for multiplicities and polarities but does not specify that the content of a message is a tuple of channel names. Kobayashi *et al.* exploit the fact that linear and non-linear channel types differ only in their multiplicity, and give a unified presentation in which every type consists of a polarity, a multiplicity and a message type. In order to be able to generalise our formalisation to type systems in which linear and non-linear types might differ in more substantial ways (particularly *session types* — see Section 11), we take a different approach. The linear and non-linear cases are defined as separate constructors for the polymorphic type `vty`, which represents the type of a variable in an environment.

```
datatype ('a,'b) vty = LinV ('a linv) | NLinV 'b
```

Ultimately we use the instantiation $(lty, nltv) vty$ where lty is the type of linear channel types and $nltv$ is the type of non-linear channel types.

The type $linv$ is a record with two fields, pos and neg , representing the positive (output) and negative (input) occurrences of a variable.

```
record 'a linv =   pos :: 'a oty   neg :: 'a oty
```

The type oty is defined by

```
datatype 'a oty = Zero | One 'a
```

and the constructors represent non-use or use of a capability. For example, the type $?^1T$ is represented in an environment by the type

```
LinV (| pos = Zero , neg = One T |).
```

For non-linear types, we drop the polarities and do not distinguish between the two ends of a channel.

The operation $addt$ (addition of types), the predicate $canaddt$ (testing definedness of addition), the predicate $unlim$ (testing whether an individual type is unlimited, which is later extended to a predicate $unlimited$ on environments), and several others, can be defined at this stage. Later, in the theory Π , we define the (mutually recursive) types

```
datatype lty = LChan ((lty, nltv) vty list)
and       nltv = NLChan ((lty, nltv) vty list)
```

and use $(lty, nltv) vty list$ as the type of an environment.

Our departure from the unified presentation of linear and non-linear types means that when formalising the typing rules we need separate rules for input on linear and non-linear channels, and similarly for output, but the extra generality of the theory $Types$ compensates for this complication in the theory Π .

Another way in which our formalisation differs from the presentation of Kobayashi *et al.* is that when defining addition of linear types, we check the disjointness condition on the polarities but do not require the message types to be equal. This makes it easier to generalise to session types, but requires us eventually to impose a global condition that the environments of typed processes be *balanced*, which means that every linear type in the environment has the same message type associated with its positive and negative ends.

A large number (over 60) of essential basic theorems about types can be proved independently of the specific definitions of linear and non-linear types. A typical example is commutativity of addition of types:

```
canaddt t u ==> addt t u = addt u t
```

The function $addt$ should be partial, but for convenience is defined as a total function in Isabelle; in the unwanted cases, its value is arbitrary. To make sure that these arbitrary values do not interfere with the properties of $addt$, the predicate $canaddt$ is used to check that arguments to $addt$ are within its domain of definition.

The types of `t` and `u` are `vty`. The proofs of these theorems require case-analysis on `t` and `u`, using the exhaustion theorem (automatically supplied by Isabelle) for the datatype `vty`; use of Isabelle's tactic `record_split_tac` to decompose goals involving variables of type `linv`; and further case-analysis using the exhaustion theorem for the datatype `oty`. Simplification is also used at each stage.

All of these theorems are proved in essentially the same way, and we have implemented a specialised tactic, `Types_tac`, for this purpose. Unfortunately, when applying case-analysis it is necessary to provide the name of the variable involved. This means that `Types_tac` only works if the theorems are stated in terms of the variables `t`, `u`, `v`, `w` or a subset of these; none of the theorems involve more than 4 variables. Furthermore, the second round of case-analysis depends on the exact names of variables which are introduced by Isabelle during the first round. The behaviour of `Types_tac`, in terms of the number of subgoals generated and the time taken by simplification, depends on the order in which variables are considered in the second round of case-analysis. In practice, `Types_tac` works well and proves all but two of the theorems in `Types`. The remaining theorems are proved by a variation of `Types_tac` which considers variables in a different order. The dependence on particular variable names, particularly internally-generated names, means that `Types_tac` is not robust and this is somewhat unsatisfactory. We are still trying to find a better way of expressing the proofs in this theory.

5 Environments

The theories `Env` and `TypedEnv` define operations on environments. An environment is a list of types, and variable indices refer to positions within an environment. `Env` defines all operations which can be expressed independently of `Types`, for example the function `typeof` which returns the type at a particular position in an environment. The operations defined by `Env` are either polymorphic operations on lists, which will later be applied to environments, or operations manipulating positions within environments, which are therefore functions on integers or lists of integers.

`TypedEnv` defines, in terms of `addt`, operations associated with addition of types and environments: `addtypetoenv`, which adds a capability to the type at a given position within an environment; `addtoenv`, which adds a list of capabilities to the types at a list of positions; and `addenv`, which adds environments. All of these operations are partial, because `addt` is partial. Each operation therefore has an associated predicate, for example `canaddenv`, which is used in the hypotheses of theorems about the partial operations. For example, the theorem that addition of environments is commutative is

```
canaddenv G H ==> addenv G H = addenv H G.
```

This theorem is proved by induction on `G` and case-analysis on `H`, using the corresponding theorem about `addt` (mentioned in Section 4).

6 The Meta Language

Rather than define a representation of pi terms directly, we use a meta language, which is a lambda calculus with constants. This approach has a number of advantages.

- Once basic properties of the meta language (e.g. injectivity of constructors) have been proved, the corresponding properties of the pi calculus are easily proved by Isabelle’s automatic tactics, and it is trivial to transfer the proofs to extensions or variations of the calculus.
- The meta language is simpler (has fewer constructors) than the pi calculus. This means that some key theorems, whose proofs are complex, are easier to prove for the meta language and lift to the pi calculus than to prove for the pi calculus directly; see Section 8.
- Both of the n -ary binding operators of the pi calculus are represented in terms of a single unary binding operator in the meta language; issues to do with variable binding are therefore concentrated into a single, relatively simple, area.

We represent variables by de Bruijn indices [1], to avoid the complications of reasoning about α -conversion and freshness of variable names in substitutions [7]. Theory `Meta` defines the following datatype, parameterised by a type of constants.

```
datatype 'a meta =
  mCon 'a                (* constant *)
| mVar var                (* variable *)
| mAbs ('a meta)         (* abstraction *)
| mApp ('a meta) ('a meta) (* application *)
| mAdd ('a meta) ('a meta) (* additive application *)
```

The type `var` is a record type with fields `index`, of type `nat`, and `tag`, whose type is an enumeration with possible values `Pos`, `Neg` or `Both`. The `index` field is the de Bruijn index of the variable occurrence. The `tag` field distinguishes between positive and negative occurrences of linear variables and occurrences of non-linear variables (which have tag `Both`). This explicit tagging of variable occurrences simplifies some proofs by making it easier to determine which end of a linear channel is being transmitted by an output. In a practical language this information could probably be obtained by type inference, and it might be possible to eliminate it from our formalisation in the future. More generally, we anticipate other uses of the `tag` field when adapting our formalisation to other type systems; for example, Pierce and Sangiorgi’s proof of type soundness for the pi calculus with subtyping [20] uses an auxiliary syntax in which variable occurrences are tagged with runtime types.

The de Bruijn index of a variable occurrence counts the number of abstractions between the occurrence and the abstraction which binds it. Indices so large that they point outside all abstractions refer to positions in a notional list of free

variables; this list is, in effect, an untyped environment for the term. For example, if the environment of free variables is taken to be u, v (the order is arbitrary), and if we pretend that `var = nat` and omit the `mVar` constructors, then the term

$$\lambda x.u(\lambda y.u(x(yv)))$$

is represented by `mAbs (mApp 1 (mAbs (mApp 2 (mApp 1 (mApp 0 3)))))`.

It is slightly misleading to think of the meta language as a lambda calculus. Because we never apply β -reduction to meta language terms, `mAbs` and `mApp` are unrelated; in particular, the order of the arguments of `mApp` is unimportant. Note the inclusion of a second “application”, called `mAdd`, which will be used when representing additive linear operators; it is convenient for multiplicative and additive operators to use different meta language constructors.

In order to support the n -ary binding operators of the pi calculus, and to represent output (which involves a list of variables), we define `mAbs` for iterated abstraction and `mApp` which applies `mApp` repeatedly to the elements of a list.

7 Substitution

Substitution is defined for the meta language in theory `Meta` and lifted to the pi calculus in theory `Pi`. In order to define substitution appropriately, we consider the key case in the proof of the subject reduction theorem. The derivation

$$\frac{\frac{\Gamma \vdash P}{\Gamma + x : !^1[\tilde{T}] + \tilde{y} : \tilde{T} \vdash x![\tilde{y}]. P} \quad \frac{\Delta, \tilde{z} : \tilde{T} \vdash Q}{\Delta + x : ?^1[\tilde{T}] \vdash x?[\tilde{z} : \tilde{T}]. Q}}{\Gamma + \Delta + x : \downarrow^1[\tilde{T}] + \tilde{y} : \tilde{T} \vdash x![\tilde{y}]. P \mid x?[\tilde{z} : \tilde{T}]. Q}$$

types the process

$$x![\tilde{y}]. P \mid x?[\tilde{z} : \tilde{T}]. Q$$

which reduces to

$$P \mid Q\{\tilde{y}/\tilde{z}\}$$

and we must therefore derive a typing for the latter process. If we have

$$\Delta + \tilde{y} : \tilde{T} \vdash Q\{\tilde{y}/\tilde{z}\}$$

then we can derive

$$\Gamma + \Delta + \tilde{y} : \tilde{T} \vdash P \mid Q\{\tilde{y}/\tilde{z}\}$$

which is sufficient; note that the linear input and output capabilities of x were used by the communication and are not needed in order to type $P \mid Q\{\tilde{y}/\tilde{z}\}$. This example shows that the variables \tilde{z} do not appear in the environment of $Q\{\tilde{y}/\tilde{z}\}$, but that the types of \tilde{z} must be added to the environment as extra capabilities for \tilde{y} . Theory `Subst` defines a function `substenv` which calculates the typed environment for a substituted term, and a predicate `oksubst` which tests definedness of this environment.

We have already noted that it is sufficient to define substitution of variables (not arbitrary terms) for variables. Furthermore, because it is only necessary to substitute for input-bound variables, we can assume that the variables being replaced are distinct from each other and from the variables replacing them; it is therefore possible to define the simultaneous substitution of a list of variables as a sequence of substitutions of single variables. This is different from other pi calculus formalisations (for example, Melham's [14]) which define simultaneous substitution directly, but given that we are using de Bruijn indices it represents a very significant simplification. Because substitution in a typed process requires a change in the environment, the variable indices throughout the process must be adjusted. This adjustment would be rather complex to define for simultaneous substitutions, but is much more straightforward for a single substitution. Similarly, proofs about simultaneous substitution use induction to extend results about single substitution to lists, and this is simpler than reasoning directly about simultaneous substitution. Reasoning about the adjustment of indices in single substitutions is facilitated by Isabelle's recently-introduced tactic `arith_tac`, which is based on a decision procedure for linear arithmetic.

8 The Typed Meta Language

Another novel feature of our formalisation is that we define a type system for the meta language, which keeps track of linear variable use. The rules (omitting tags, and presented informally) are shown in Figure 1.

$$\begin{array}{c}
\frac{\Gamma \text{ unlimited}}{\Gamma + v : \text{LinV } (| \text{ pos} = \text{One } T, \text{ neg} = \text{Zero } |) \vdash_m \text{mVar } v} \\
\frac{\Gamma \text{ unlimited}}{\Gamma + v : \text{LinV } (| \text{ pos} = \text{Zero}, \text{ neg} = \text{One } T |) \vdash_m \text{mVar } v} \\
\frac{\Gamma \text{ unlimited}}{\Gamma + v : \text{NLinV } T \vdash_m \text{mVar } v} \\
\frac{\Gamma \text{ unlimited}}{\Gamma \vdash_m \text{mCon } c} \\
\frac{\Gamma, x : T \vdash_m t}{\Gamma \vdash_m \text{mAbs } t} \\
\frac{\Gamma \vdash_m t \quad \Delta \vdash_m u}{\Gamma + \Delta \vdash_m \text{mApp } t u} \\
\frac{\Gamma \vdash_m t \quad \Gamma \vdash_m u}{\Gamma \vdash_m \text{mAdd } t u}
\end{array}$$

Fig. 1. Typing rules for the meta language.

A few technical theorems are proved at this stage, for example:

If $\Gamma \vdash_m p$ and $\Delta \vdash_m p$ and $\Gamma + \Delta$ is defined and Γ is unlimited then Δ is unlimited.

The proof of this theorem is simpler for the typed meta language than it would be for the typed pi calculus, because the meta language has fewer constructors. In the theory `Pi` we prove that every typing rule for the pi calculus is a

derived rule for the meta language type system. Hence this theorem, and others like it, can be lifted very easily to the pi calculus level, and extended easily to variations of the pi calculus. This particular theorem is used in the proof that structural congruence preserves typability (Section 10).

This technique only applies to theorems whose conclusions do not mention the typing relation. Correct typing of (the representation of) a pi calculus term as a meta language term does not imply correct typing of the pi calculus term — the pi calculus typing rules have extra hypotheses which are not captured by the meta language typing rules. Nevertheless, we have found the introduction of the meta language type system to be a useful structuring technique which very significantly simplifies the proofs of the theorems to which it is applicable.

9 Formalising the Pi Calculus

The theory `Pi` defines specific linear and non-linear channel types and a specialised type of environments, as mentioned in Section 4. A specific type `con` of constants for the meta language is defined, and representations of pi calculus terms are defined as meta language terms of type `pi = con meta`. For example,

```
OUT x n xs p == mAPP (mApp (mCon (Out n)) p) (map mVar (x # xs))
```

represents $x![xs].p$ where $length(xs) = n$.

The structural congruence relation, the reduction relation and the typing relation are all defined inductively. The de Bruijn representation affects some rules: for example, the scope extrusion rule for structural congruence (Section 2) becomes `(SCONG is an inductively-defined subset of pi * pi, formalising the structural congruence relation, so that (p,q):SCONG means p ≡ q)`

```
(PAR p (NEW n ts q) , NEW n ts (PAR (lift 0 n p) q)) : SCONG
```

where `lift 0 n p` adjusts the indices of the free variables of `p` to take account of the insertion of `n` variables at position 0 in the environment. The definition of `lift` guarantees that the first `n` variables in the environment, corresponding to \tilde{x} , do not occur in `lift 0 n p`, so the condition on free variables disappears.

As indicated in Section 7, if $\Gamma \vdash P$ and $P \longrightarrow Q$ by a communication on a linear channel, then Q is typed in an environment in which the polarity of that channel has been reduced to \downarrow . It is therefore necessary to label a reduction with the name of the channel involved (or τ if the channel is hidden by an enclosing ν), represented by type `redch`. The reduction $P \xrightarrow{x} Q$ is formalised by `(P , x , Q) : RED` where `RED` is an inductively defined subset of `pi * redch * pi`.

Defining pi terms via a meta language rather than directly as a datatype means that distinctness and injectivity theorems need to be proved explicitly for the pi calculus constructors. The number of such theorems is fairly large (the square of the number of constructors), but they are all proved by Isabelle's automatic tactics.

Theorems about substitution on pi terms are also proved automatically, for example (note the addition of `n` due to changing environments):

$(\text{NEW } n \text{ ts } p)[x//i] = \text{NEW } n \text{ ts } p[x+n // i+n]$.

Similarly, theorems about `lift` on pi terms are proved automatically.

Theorems stating that the pi typing rules are derived rules for the typed meta language are proved, as mentioned in Section 8. Each theorem is simply a pi typing rule with the pi typing relation replaced by the meta typing relation. The proofs are short, the only complexity being some explicit use of theorems about environments.

10 Type Soundness

The key to proving soundness of the type system is the *subject reduction theorem*, which states that reduction of a typed process yields a typed process. We prove

$[| G \vdash p ; (p, x, q) : \text{RED} ; \text{balanced } G |] \implies \text{usechan } x \ G \vdash q$

by induction on the reduction relation, `RED`. As discussed in Section 7, `q` is typed not in `G` but in an environment in which the capability of the channel `x` has been used, if its type is linear.

As we also saw in Section 7, we need to prove a *substitution lemma*:

$[| G \vdash p ; \text{oksubst } x \ i \ G |] \implies \text{substenv } x \ i \ G \vdash p[x//i]$

for substitutions of single variables and extend it by induction to substitutions of lists of variables. Here we use the induction principle derived from the inductive definition of the typing relation. The proof breaks down into a number of cases, one for each pi calculus constructor. In order to make it easier to extend the language later, we prove a lemma for each case and then combine them with a “driver” proof which simply applies the induction rule to generate the cases and then applies the appropriate lemma to each case. Once the proof is structured in this way, it is easy to see how to extend it if new constructors are added. All of our proofs about the typing relation are similarly modularised.

Because structural congruence can be used when deriving reductions, we prove that structural congruence preserves typability:

$[| G \vdash p ; (p, q) : \text{SCONG} |] \implies G \vdash q$.

The proof is by induction on the definition of `SCONG`, and as with the inductive proofs about `|-`, we structure the proof by extracting the cases as separate lemmas. The congruence cases are almost trivial and are all proved by a simple specialised tactic, `SCONG_tac`, which uses Isabelle’s classical reasoner to search for a proof involving the introduction and elimination rules arising from the inductive definition of the typing relation. Most of the other cases are straightforward, involving application of the typing rules and some reasoning about addition of environments. Some of the cases dealing with replication and restriction are more complex, requiring (for replication) the fact about unlimited environments mentioned in Section 8 and (for restriction) the following *weakening lemma*:

$[| G \vdash p ; \text{length } ts = n ; \text{unlimited } ts ; m \leq \text{length } G |] \implies$
 $\text{ins } ts \ m \ G \vdash \text{lift } m \ n \ p$

in which $\text{ins } ts \ m \ G$ is the environment obtained by inserting the types ts into G at position m . Another form of weakening adds capabilities to types which are already in the environment. We prove

$[| G \vdash p ; \text{unlimited } H ; \text{canaddenv } G \ H |] \implies \text{addenv } G \ H \vdash p$

by induction on the typing relation, again structuring the proof into a lemma for each case.

The subject reduction theorem states that reductions of typed processes yield typed processes, but because reduction is not defined for communications with an arity mismatch, we need to prove that these incorrect communications cannot occur in typed processes. Following Kobayashi *et al.* we prove that if a typed process contains a term $x![\tilde{z}].P \mid x?[\tilde{y} : \tilde{T}].Q$ then $\text{length}(\tilde{y}) = \text{length}(\tilde{z})$ and the type of x (in the environment or in an enclosing restriction) has polarity \uparrow . We also prove their theorems about correct use of linear and non-linear channels, for example that if a typed process contains a term $x![\tilde{z}].P \mid x![\tilde{z}].Q$ then the type of x is non-linear.

11 Extension to Session Types

Honda *et al.* [10, 24] have proposed *session types*, which allow successive communications on a channel to carry different message types. Branching is also allowed, so that the transmission of a particular value (from a finite range) can determine the type of the subsequent interaction. Consider, for example, a server for mathematical operations, which initially offers a choice between equality testing and negation. A client must choose an operation and then send the appropriate number of arguments to the server, which responds by sending back the result. All communications take place on a single session channel called x ; the type of the server side of this channel is

$$S = \&(\text{eq} : ?[\text{int}] . ?[\text{int}] . ![\text{bool}] . \text{end}, \text{neg} : ?[\text{int}] . ![\text{int}] . \text{end}).$$

The author and Malcolm Hole [5] have defined a notion of subtyping for session types, proposing a language whose type system allows the description of backward-compatible upgrades to client-server systems. Ultimately we aim to formalise the type system of this language.

To avoid runtime type errors, a session channel must be shared by exactly two processes, one at each end. Within each process, however, the channel must be used for as many communications as are specified by its type. Session types therefore require an interesting mixture of linear and non-linear (but bounded) control and this will require a generalisation of our present theory **Types**. Addition of a capability to a session type should mean prefixing a new communication to the existing type, for example extending $![\text{bool}] . \text{end}$ to $?[\text{int}] . ![\text{bool}] . \text{end}$.

Using Pierce and Sangiorgi's [20] technique, proving type soundness for the system with subtyping requires the definition of an auxiliary language in which variable occurrences are tagged with runtime types. Our formalisation will have to define two different, although related, languages, and our structured approach should make it easy to achieve this, especially as we have already included support for tagging of variable occurrences.

12 Conclusions

We have formalised a linear type system for the pi calculus, and a proof of its soundness, as a step towards a formalisation of a more complex type system incorporating session types and subtyping.

Our work differs from other formalisations of the pi calculus in several significant ways. One is that we concentrate on reasoning about the reduction relation and type systems, including systems with linear features, rather than labelled transition systems and bisimulation. More importantly, mindful of our desire to extend this work to a more complex language, we have aimed to develop a methodology and general framework for the formalisation of type systems for languages based on the pi calculus, rather than simply to formalise a particular type system. The use of a typed meta language (Sections 6 and 8), the development of a generic theory of linear and non-linear types (Section 4) and environments (Section 5), and the modular structure of the proofs about the pi calculus typing relation (Section 10), are all intended to facilitate the transfer of our techniques to other languages.

References

- [1] N. G. deBruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
- [2] J. Despeyroux. A higher-order specification of the pi-calculus. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, 2000.
- [3] C. Dubois. Proving ML type soundness within Coq. In M. Aagaard and J. Harrison, editors, *Proceedings of TPHOLs2000, the 13th International Conference on Theorem Proving in Higher Order Logics*, LNCS, pages 126–144. Springer, 2000.
- [4] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 2001.
- [5] S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In S. D. Swierstra, editor, *ESOP'99: Proceedings of the European Symposium on Programming Languages and Systems*, number 1576 in Lecture Notes in Computer Science, pages 74–90. Springer-Verlag, 1999.
- [6] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
- [7] A. D. Gordon and T. Melham. Five axioms of alpha conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.
- [8] L. Henry-Gréard. Proof of the subject reduction property for a π -calculus in COQ. Technical Report 3698, INRIA Sophia-Antipolis, May 1999.

- [9] D. Hirschhoff. A full formalisation of π -calculus theory in the calculus of constructions. In *Proceedings of the Tenth International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.
- [10] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming*, *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [11] F. Honsell, M. Miculan, and I. Scagnetto. π -calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.
- [12] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, September 1999.
- [13] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3), 1999.
- [14] T. F. Melham. A mechanized theory of the π -calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.
- [15] R. Milner. The polyadic π -calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.
- [16] R. Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
- [17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.
- [18] L. C. Paulson. *Isabelle — A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [19] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Programming Language Design and Implementation*. ACM Press, 1988.
- [20] B. C. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.
- [21] C. Röckl. A first-order syntax for the pi-calculus in Isabelle/HOL using permutations. Technical report, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 2001.
- [22] C. Röckl, D. Hirschhoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, number 2030 in *Lecture Notes in Computer Science*, pages 364–378. Springer-Verlag, 2001.
- [23] D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.
- [24] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [25] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.
- [26] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.