A Concurrent Programming Language with Refined Session Types

Juliana Franco(✉) and Vasco Thudichum Vasconcelos

LaSIGE, Faculdade De Ciências, Universidade De Lisboa, Lisboa, Portugal
jfranco@lasige.di.fc.ul.pt

Abstract. We present SePi, a concurrent programming language based on the monadic pi-calculus, where interaction is governed by linearly refined session types. On top of the core calculus and type system, and in order to facilitate programming, we introduce a number of abbreviations and derived constructs. This paper provides a brief introduction to the language.

1 Introduction

Session types [12] are by now a well-established methodology for typed, message-passing concurrent computations. By assigning session types to communication channels, and by checking programs against session type systems, a number of important program properties can be established, including the absence of races in channel manipulation operations, and the guarantee that channels are used as prescribed by their types. As a simple example, a type of the form $!\mathsf{string}.!\mathsf{integer}.\mathsf{end}$ describes a channel end on which processes may first output a string, then output an integer value, after which the channel provides no further interaction. The process holding the other end of the channel must first input a string, then an integer, as described by the complementary (or dual) type, $?\mathsf{string}.?\mathsf{integer}.\mathsf{end}$. If the string denotes a credit card number and the integer value the amount to be charged to the credit card, then we may further *refine* the type by requiring that the capability to charge the credit card has been offered, as in $?\mathsf{ccard}{:}\mathsf{string}.?\mathsf{amount}{:}\{\mathsf{x}{:}\mathsf{integer}|\mathsf{charge}(\mathsf{ccard},\mathsf{x})\}.\mathsf{end}$. The most common approach to handle refinement types is classical first-order logic which is certainly sufficient for many purposes but cannot treat formulae as resources. In particular it cannot guarantee that the credit card is charged with the given amount only once.

SePi is an exercise in the design and implementation of a concurrent programming language solely based on the message passing mechanism of the pi calculus [16], where process interaction is governed by (linearly refined) session types. SePi allows to explore the practical applicability of recent work on session-based type systems [1,25], as well as to provide a tool where new program idioms and type developments may be tested and eventually incorporated. In this respect, SePi shares its goal with Pict [19] and TyCO [22].

S. Counsell and M. Núñez (Eds.): SEFM 2013 Collocated Workshops, LNCS 8368, pp. 15–28, 2014.
DOI: 10.1007/978-3-319-05032-4_2, © Springer International Publishing Switzerland 2014

The SePi core language is the monadic synchronous pi-calculus [16] with replication rather than recursion [15], labelled choice [12], and with assume/assert primitives [1]. On top of this core we provide a few derived constructs aiming at facilitating code development. The current version of the language includes support for mutually recursive process definitions and type declarations, for polyadic message passing and session initiation, a **dualof** operator on types, and an abbreviation for shared types. The type system of SePi is that of linearly refined session types [1], the algorithmic rules for the refinement-free type language are adapted from [25], and those for refinements are described in this paper.

SePi is currently implemented as an Eclipse plug-in, allowing code development with the usual advantages of an IDE, such as syntax highlighting, syntactic and semantic validation, code completion and refactoring. It further includes a simple interpreter based on Turner's abstract machine [21]. There is also a command line alternative, in the form of a jar file. Installation details and examples can be found at http://gloss.di.fc.ul.pt/sepi.

The rest of this paper is structured as follows. The next Section reviews related work. Section 3 briefly introduces SePi based on a running example. Section 4 presents a few technical aspects of the language. Section 5 concludes the paper, pointing possible future language extensions.

2 Related Work

This section briefly reviews programming language *implementations* either based on the pi-calculus or that incorporate session types.

There are a few programming languages based on the pi-calculus, but none incorporate session types. Pict [19] is a language in the ML-tradition, featuring labelled records, higher-order polymorphism, recursive types and subtyping. Similarly to the SePi approach, Pict builds on a tiny core (a variant of the asynchronous pi-calculus [3,11]) by adding a few derived constructs. TyCO [23] is another language based on a variant of the asynchronous pi-calculus, featuring labelled messages (atomic select/output) and labelled receptors (atomic branch/input) [22], predicative polymorphism and full type inference. In turn, SePi is based on the monadic synchronous pi-calculus with labelled choice [12], explicitly typed and equipped with refined session types [1]. Polymorphism and subtyping are absent from the current version of SePi.

On the other hand, we find programming languages that feature session types or variants of these, but are based on paradigms other than the pi-calculus. For functional languages, we have those that take advantage of the rich system of Haskell, monads in particular, and those based on ML. Neubauer and Thiemann implemented session types on Haskell using explicit continuation passing [17]. Sackman and Eisenbach improve this work, augmenting the expressive power of the language [20]. Given that session types are encoded, the Haskell code for session-based programs can be daunting. SePi works directly with session types, thus hopefully leading to readable programs. Bhargavan et al. [2] present a ML-like language for specifying multiparty sessions [13] for cryptographic protocols, with integrity and secrecy support.

For object-oriented languages, Fähndrich et al. developed Sing# [6], a variant of C# that supports message-based communication via shared-memory where session types are used to describe communication patterns. Hu et al. introduced SJ [14], an extension of Java with specific syntax for session types and structured communication operations. Based on a work by Gay et al. [7], Bica [4] is an extension of the Java 5 compiler that checks conventional Java source code against session type specifications for classes. Type specifications, included in Java annotations, describe the order by which methods in classes should be called, as well as the tests clients must perform on results from method calls. Following a similar approach, but using session types with lin/un annotations [25], Mool [5] is a minimal object based language.

Finally, for imperative languages, Ng et al. developed Session C [18], a multiparty session-based programming environment for the C programming language and its runtime libraries [18]. Also using the theory of multiparty session types, we have the Scribble framework presented by Honda et al. [10], that supports bindings for several high-level languages such as ML, Java, Python, C# or C++, and whose purpose is to provide a formal and intuitive language and tools to specify communication protocols and their implementations. Neither of the works discussed above feature any form of refinement types, linear or classical.

3 A Gentle Introduction to the SePi Language

This section introduces the SePi language, its syntax, type system and operational semantics. The presentation is intentionally informal. Technical details can be found on the theoretical work the language builds upon, namely [25] for the base language and [1] for refinements.

Our running example is based on the online petition service [24] and on the online store [1]. An Online Donation Server manages donation campaigns. Clients seeking to start a donation campaign for a given cause begin by setting up a session with the server. The session is conducted on a channel on which the campaign related data is provided. The same channel may then be disseminated and used by different benefactors for the purpose of collecting the actual donations. Parties donating for some cause do so by providing a credit card number and the amount to be charged to the card. The type system makes sure that the exact amount specified by the donor is charged, and that the card is charged exactly once.

SePi is about message passing on bi-directional synchronous channels. Each channel is described by two end points. Processes may write on one end or else read from the other end, at each particular location in a program. Channels are governed by types that describe the sequence of messages a channel may carry. We start with *input/output types*. A type of the form `!integer.end` describes a channel end where processes may write an integer value, after which the channel offers no further interaction. Similarly, a type `?integer.end` describes a channel end from which processes may read an integer value, after which the channel offers no further interaction.

To *create a channel* of the above type one writes

```
new w r: !integer.end
```

Such a declaration introduces two new program variables: w of type !**integer**.**end**, and r of type ?**integer**.**end**. A semantically equivalent declaration is **new** r w: ?**integer**.**end** . To *write* the integer value 2013 on the newly created channel, one uses w!2013. To *read* from the channel and let program variable x denote the value read, one writes r?x. For the purpose of printing integer values on the console, SePi provides the primitive channels printInteger and printIntegerLn , and similarly for the remaining base types: **boolean** and **string**. The Ln versions issue a newline after printing the value. Code such as channel writing or reading can be composed by *prefixing* via the dot notation. To read an integer value and then to print it, one writes r?x. printInteger !x. To run two processes in *parallel* one uses the vertical bar notation. Putting everything together one obtains our first complete program, composed of a channel declaration and two processes running in parallel while sharing the channel.

```
new w r: !integer.end
w!2013 | r?x.printInteger!x
```

Running such a program would produce 2013 on the console, after which the program terminates.

We now move on to *choice types*. The donation server allows clients to setup donation campaigns piece-wise. The required information (title, description, due date, etc.) may be introduced in any order, possibly more than once each. Once satisfied, the client “presses the commit” button. A channel end that allows a writer to *select* either the setDate option or the commit option is written as:

```
+{setDate:end, commit:end}
```

Conversely, a channel end that provides a menu composed of the two same choices can be written as &{setDate:**end**, commit:**end**}. To select the setDate option on a + channel end we write w **select** setDate. Conversely to branch on a & channel end one may write **case** r **of** setDate → ... commit → ... . Putting everything together one obtains the following process.

```
new w r: +{setDate:end, commit:end}
w select setDate |
case r of setDate → printString!"Got setDate"
          commit  → printString!"Got commit"
```

We have seen that types are composed by prefixing, using the dot notation: !**integer**.**end** means write an integer and then go on as **end**. We can compose the output and the select type we have seen above, so that the output of an integer is required after the setDate choice is taken. We leave to the reader composing the two programs above so that it interacts correctly on a channel whose client end is of type +{setDate:!**integer**.**end**, commit:**end**}.

The problem with this type is that it does not reflect the idea of “uploading the campaign information until satisfied, and then press the commit button”. All a client can do is *either* set the date *or else* commit. What we would like to say is that after the setDate choice is taken the whole menu is again available. For this we require a *recursive* type of the form:

```
rec a.+{setDate:!integer.a, commit:end}
```

A client w may now upload the date two times before committing:

```
w select setDate. w!2012.
w select setDate. w!2013. w select commit
```

The donation server, when governed by type etDate:?integer.a, commit:end**rec** a.& s, needs to continuously offer the setDate and commit options. Such behaviour cannot be achieved with a finite composition of the primitives we have seen so far. We need some form unbounded behaviour, which SePi provides in the form a **def** process. The setup process below is the part of the donation server responsible for downloading the campaign information. To simplify the example, only the due date is considered and even this information, x, is immediately discarded. We will see that setup is a form of an input process that survives interaction, thus justifying its invocation with the exact same syntax as message sending: setup!r.

```
def setup r: rec a.&{setDate:?integer.a, commit:end} =
    case r of setDate → r?x. setup!r
              commit  → ...
```

Process definition, **def**, is the second form of declaration in SePi (the first is **new**). There is yet a third kind of declaration (rather, an abbreviation): **type**. Introducing the name Donation for the above recursive type, one may write:

```
type Donation = +{setDate: !integer.Donation, commit: end}
```

thus foregoing the explicit use of the **rec** type constructor. Type, process and channel declarations may be mutually recursive. Keywords **type**, **def**, and **new** introduce a series of equations that are elaborated by the compiler, as described in Sect. 4.

There is a further handy abbreviation. Session types tend to be quite long; if a channel's end point is of type **rec** a.+{setDate:!**integer**.a, commit:**end**}, the other end is of type **rec** a.&{setDate:?**integer**.a, commit:**end**}. In this case we say that one type is *dual* to the other, a notion central to session types. Given that we abbreviated the first type to Donation, the second can be abbreviated to **dualof** Donation. Putting every together we obtain the following process.

```
type Donation = +{setDate: !integer.Donation, commit: ...}
def setup r: dualof Donation =
    case r of setDate → r?x. setup!r
              commit  → ...
new w r: Donation          // the donation channel
w select setDate. w!2012. w select setDate. w!2013.
    w select commit |      // a client
setup!r                    // a server
```

Continuing with the example, after setup comes the promotion phase. Here the donation channel is used to collect donations from benefactors. Benefactors donate to a cause by providing a credit card number and the amount to be charged to the card. So we rewrite the donation type to:

```
type Donation = +{setDate:!integer.Donation, commit:Promotion}
type CreditCard = string
```

How does type Promotion look like? If we make it !CreditCard.!integer.end, then the server accepts a single donation. Clearly undesirable. If we choose rec a.!CreditCard.!integer.a, then we accept an infinite number of donations. And this is undesirable for two reasons: (a) regrettably, no campaign will ever receive an infinite number of donations, and (b) all these donations would have to be issued from the same thread (a process without parallel composition), one after the other. The first problem can be easily circumvented with a rec-choice combination, as in type Donation. The root of the second problem lies in the fact that types are *linear* by default, meaning that each channel end can be known, at any given point in the program, by exactly one thread. And this goes against the idea of disseminating the channel in such a way that any party may individually donate, by just knowing the channel. We need a means to say that channel ends can be shared by more than one process. Towards this end, we label *each prefix* as either **un** or **lin**. Shared types are qualified with **un** (for unrestricted); linear types with **lin**. It turns out that the **lin** qualifier is optional. For example, !**integer**.**end** abbreviates **lin** !**integer**.**end**.

The type system keeps track of how many threads know a channel end: if **lin** then exactly one, if **un** then zero or more. Linear channels are exempt from races: we do not want two threads competing to set up a donation campaign. Shared channels are prone to races: we do want many (as many as possible) simultaneous benefactors carrying out their donations. Care must however be exerted when using shared channels. Imagine that type Promotion looks like **rec** a.**un**!CreditCard.**un**!**integer**.a, and that we have two donors trying to interact with the server,

```
w!"2345".w!500 | w!"1324".w!2000 | r?x.r?y...
```

Further imagine that the first donor wins the race, and exchanges message "2345". We are left with a process of the form w!500 | w!"1324".w!2000 | r?y... , where the value transmitted on the next message exchange can be an integer value (500) or a string ("1324"), a situation clearly undesirable. To circumvent this situation we pass the two values in a single message, by making w of type **rec** a.**un**!(CreditCard,**integer**).a. This pattern, **rec** a.**un**!T.a, is so common that we provide an abbreviation for it: ∗!T, and similarly for input. So here is the new type for Promotion.

```
type Promotion = *!(CreditCard,integer)
```

Now a client can donate twice (in parallel); it may also pass the channel to all its acquaintances so that they may donate and/or further disseminate the channel. In the process below, notice the parallel composition operator enclosed in braces when used within a process.

```
w select setDate. w!2014. w select commit. {
    w!("2345", 500) | w!("1324", 2000) | acquaintance!w
}
```

The ability to define types that "start" as linear (e.g. Donation) and end up as unrestricted (Promotion) was introduced in [25].

So far our example is composed of one server and one client. What if we require more than one client (the plausible scenario for an online system) or more than one server (perhaps for load balancing)? If we add a second client, in parallel with the above code for the server and the client, the program does not compile anymore: there is a race between the two clients for the linear channel end w. On the one hand we have seen that the donation channel must be linear; on the other hand we want a donation server reading on a well-known, public, **un**, channel. We start by installing the server on a channel end of type ∗?Donation, and disseminate the client end of the channel (of type **dualof** ∗!Donation, that is ∗?Donation). Our main program with two clients looks as follows.

```
new c s: *?Donation      // create an Online Donation channel
donationServer!s |       // send one end to the Donation Server
client1!c | client2!c    // let the whole world know the other
```

To this pattern—create a channel, send one end to the server, keep the other—we call *session initiation*. We found it so common that we introduced an abbreviation for it. The above three lines of code can be replaced with the following process.

```
donationServer!(new c: *?Donation). { client1!c | client2!c }
```

Now the first output introduces a binding (for program variable c), hence we cannot use parallel composition anymore. Instead we use prefix. One of the advantages of the session initiation abbreviation is that it spares the programmer from coming up with two different identifiers; that for the server end becomes implicit. Notice however that, in a session initiation process of the form x!(**new** y:T).P the actual end point that is sent is of type **dualof** T.

We now concentrate on how the donation server charges credit cards. In general, merchants cannot directly charge credit cards. As such our donation server forwards the transaction details (the credit card number and amount to be charged) to the credit card issuer (a bank, for example). Assume the following definition for a bank: **def** bank (ccard: CreditCard, amount: **integer**). Well behaved servers receive the data and forward it to the bank:

```
r?(ccard, amount). bank!(ccard, amount)
```

Not so honest servers may try to charge a different amount (perhaps a hidden tax),

```
r?(ccard, amount). bank!(ccard, amount+10)
```

or to charge the right amount, but twice.

```
r?(ccard, amount).{bank!(ccard, amount)|bank!(ccard, amount)}
```

While types cannot constitute a general panacea for fraudulent merchants, the situation can be improved. The idea is that the bank is not interested in arbitrary (ccard,amount) pairs but on pairs for which a charge (ccard,amount) capability has been granted. We then *refine* the type of the amount in the bank's signature. We are now interested on amounts x of type integer for which the predicate charge (ccard,x) holds, that is, parameter amount becomes of type

```
{x: integer | charge(ccard ,x)}
```

The capability of charging a given amount on a specific credit card is usually granted by the benefactor, by *assuming* an instance of the charge predicate, as in:

```
assume charge("2345" , 500) | w!("2345" , 500)
```

The bank, in turn, makes sure that the transaction details were granted by the client, by *asserting* the same predicate:

```
def bank (ccard: CreditCard ,
          amount: {x: integer | charge(ccard ,x)}) =
    assert charge(ccard , amount)...
```

Assumptions and assertions have no operational significance on well-typed programs. At the type system level, assumptions and assertions are treated *linearly*: for each asserted predicate there must be exactly one assumed, and conversely. In this way formulae are treated as resources: they are introduced in the type system via **assume** processes, passed around in refinement types, and consumed via **assert** processes. As such, the code for servers that try to charge twice the right amount (see above) does not type check, for the bank's "second" **assert** is not matched by any assumption. The code for servers that try to charge a different amount (see above) does not type check either. In this case the benefactor's assumption charge("2345", 500) would never be asserted, whereas the bank's assertion charge("2345", 510) would not have a corresponding assumption. Linearity also means that code for banks that forget to **assert** charge(ccard, amount) does not type check. We leave as an exercise writing a typeful server code that charges an amount different from that stated (and assumed) by the benefactor (or that charges twice the right amount), by careful manipulation of **assume**/**assert** *in the server code.*

Benefactors that wish to be charged twice, may issue two separate assumptions or join them on a single formulae, as in the code below.

```
assume charge("2345",500)*charge("2345",500) |
w!("2345",500) | w!("2345",500)
```

Likewise, multiple assertions can be conjoined in one, via the tensor (∗) formula constructor [9].

4 Technical Aspects of the Language

SePi is based on the synchronous monadic pi calculus (as in [25]) extended with **assert** and **assume** primitives (inspired by [1]). On top of this core calculus we added a few derived constructs. This section briefly describes the core language, the derived constructs in the SePi language and the type checking system.

The *core language* includes syntactic categories for formulae A, types T, values v, expressions e, and processes P. *Formulae* in the current version of the language are built from uninterpreted predicates (over values only), tensor ($*$) and **unit**. At the *type* level we have base types (**integer**, **boolean**, **string**, and **end**), prefix types (namely, input q?x:T.U, output q!x:T.U, branching $\mathsf{q\&\{l_1{:}T_1 ,..., l_n{:}T_n\}}$ and selection $\mathsf{q{+}\{l_1{:}T_1 ,..., l_n{:}T_n\}}$, where q is either **lin** or **un**), recursion (**rec** a.T and a) and refinement types ({x:T|A}). Prefix types are labelled with an optional identifier x that may be referred to in the continuation type (e.g., !x:**integer**.!{y: **integer**|p(x,y)}.**end**).

Values in SePi are program variables (standing for channel ends), as well as integer, boolean and string constants. At the level of *expressions* SePi includes the familiar operators on integer and boolean values. For *processes* we have channel creation (**new** $\mathsf{x_1\, y_1 : T_1}$ ... **new** $\mathsf{x_n\, y_n : T_n}$ P), prefix processes (monadic input, replicated x*?y.P or use-once x?y.P, monadic output x!e.P, selection x **select** l.P, and branching processes **case** x **of** $\mathsf{l_1 {\rightarrow} P_1}$ ... $\mathsf{l_n {\rightarrow} P_n}$), conditional **if** e **then** P **else** Q, n-ary parallel composition($\{\mathsf{P_1}$ |...| $\mathsf{P_n}\}$), **assume** A and **assert** A.P. Mutually recursive channel creation **new** $\mathsf{x_1\, y_1 : T_1}$ **new** $\mathsf{x_2\, y_2 : T_2}$ allows for channel $\mathsf{x_1}$ to occur in type $\mathsf{T_2}$, and for $\mathsf{x_2}$ to occur in type $\mathsf{T_1}$.

Derived constructs at the *type* level include support for polyadic message passing ($\mathsf{q!(y_1 : T_1 ,... y_n : T_n).U}$ and $\mathsf{q?(y_1 : T_1 ,... y_n : T_n).U}$), the star syntax for unrestricted types ($*$?T, $*$!T, $*\&\{\mathsf{l_1 ,..., l_n}\}$, and $*{+}\{\mathsf{l_1 ,..., l_n}\}$), and the **dualof** type operator. Furthermore, the **lin** qualifier is optional.

Derived constructs at the *process* level include support for polyadic message passing ($\mathsf{x!(e_1 ,..., e_n).P}$ and $\mathsf{x?(y_1 ,..., y_n).P}$) and for session initiation (x !(..., **new** y:T ,...) .P). Furthermore the empty parallel composition is optional when used in the continuation of a prefix process (x!e abbreviates x!e.{}).

Finally, there is one derived construct that mixes types and processes: mutually recursive declarations of the form $\mathsf{D_1 ... D_n\, P}$, where each declaration $\mathsf{D_i}$ is either a channel creation **new** x y:T, a process definition **def** $\mathsf{x(y_1 : T_1 ,..., y_n : T_n) = P}$, or a type abbreviation **type** a = T.

We now discuss the derived constructors in SePi, starting with those related to types. A type of the form $*$?T is expanded into **rec** b.**un**?T.b for b a fresh type variable, and similarly for output, branching and selection. The **dualof** type operator produces a new type where input ? is replaced by output !, branching & is replaced by selection +, and conversely in both cases. All other type constructors remain unchanged (except for **rec**). We use the co-inductive definition of Gay and Hole [8], extended to refinement types in the natural way.

In order to simulate interference-free polyadic message passing on shared (**un**) channels, we use a standard encoding for the send and receive operations (cf. [16,25]). For example, the pair-type (ccard: CreditCard, amount: {x: **integer**| charge(ccard,x)}) in the signature of the bank definition (Sect. 3) is equivalent to the refined linear session type

```
lin ?c: CreditCard . lin ?{x: integer | charge(c,x)}.end
```

where the prefix ?CreditCard is labelled with identifier c, so that it may be referred to in the continuation, namely in the predicate charge(c,x) for the amount to be charged. On the process side, the output process b!("2345",500).P abbreviates

```
new r w: lin?c:CreditCard.lin?{x:integer|charge(c,x)}.end
b!r.w!"2345".w!500.P
```

and the input process b?(x,y).P abbreviates

```
b?z.z?x.z?y.P
```

All process constructors in the core language were introduced in Sect. 3, except for *replication*. A replicated input process behaves as an input process, except that it survives message reception. We use ? for a linear input and *? for a replicated input. For example:

```
new w r: *!integer
w!2013 | r*?x.printInteger!x | w!2014
```

prints two integer values, while

```
new w r: *!integer
w!2013 | r?x.printInteger!x | w!2014
```

will print one only.

A *process definition* is expanded into a channel creation followed by a replicated input process. Each declaration of the form **def** p x:T = P introduces a new channel, as in **new** p p': *!T where p' is a fresh identifier, in the scope of which, we add a replicated process of the form p'*?x.P, in parallel with the rest of the program. Process definitions obviate in most cases the direct usage of replicated input processes, hiding one of the channel ends (p'), thus simplifying code development. They are also amenable to an optimisation in code interpretation [21].

Session initiation is discussed in Sect. 3. In general, a process x !(..., **new** y :T ,...) .P is expanded into a process of the form **new** z_1 z_2:T x !(..., z_2 ,...) .P', where variables z_1 and z_2 are fresh, and P' is obtained from P by replacing (free) occurrences of y by z_1. This substitution is also applied to the arguments of the output process to the right of the **new**. Fresh variables prevent the free variable capture that would occur in process x!(**new** x:**end**) or y!(x,**new** x:**end**). Our experience shows that process definition and session initiation account for the vast majority of channel creation, effectively dispensing the explicit declaration of one channel end.

A sequence of declarations followed by a process is a SePi process. Declarations may be mutually recursive. Below is an example that, when run, prints an infinite sequence of alternating **true**/**false** values. Notice the mutually recursive process (p and q) and type (T and U) definitions. Further notice that type T depends on process p, which depends on channel r, which depends on type T again.

```
type T = *!(boolean,{y:U | a(y,p)})
type U = dualof T
def p b: boolean = { assume a(r,p) | w!(not b,r). q!() }
def q () = r?(x,y). assert a(y,p). printBoolean!b. p!x
new w r: T
p!false | q!()
```

Declarations are elaborated in a few steps. In the first step, type names, channel names, and process names (which are after all channel names) are collected. This information allows us to check type formation (essentially that types do not contain free type and program variables). In the second pass, we check type formation and solve the system of equations. Systems of type equations are guaranteed to be solvable due to the presence of recursive types in the syntax of the language and to the fact that types are required to be contractive.[1] We defer to the next phase the elaboration of the **dualof** operator. For example, the solution to the system of equations above is T = **rec** u.!(**boolean** ,{y:**dualof** u | a(y,p)}).u and U = **dualof** T.[2] The third step expands the occurrences of **dualof** (co-inductively, as explained above) to obtain:

```
T = rec u.un!(boolean,
                   {y:rec v.un?(boolean,{z:v|a(z,p)}).u|a(y,p)}).u
U = rec u.un?(boolean,
                   {y:rec v.un?(boolean,{z:v|a(z,p)}).u|a(y,p)}).u
```

At this point all types are resolved. The fourth step adds to the typing context entries for new channels (in **new** and **def** declarations) with the appropriate types. Finally, the last pass checks the replicated processes obtained from process definitions. Translating the example above into the core language yields the process below.

```
new p p': rec u.un!boolean.u
new q q': rec v.un!().v
new w r: T
p'*?b. { assume a(r,p) | w!(not b,r). q!() } |s
q'*?(). r?(b,x). assert a(x,p). printBoolean!b. p!b |
p!false | q!()
```

The type system of the SePi language is decidable. The algorithmic rules are those in [25], with minor adaptations in the rules for replicated input and **case** processes. Algorithmic typing systems crucially rely on the decidability of type equivalence. Type equivalence for the non-refined language is decidable [25]. Type equivalence for SePi is also decidable thanks to the extremely simple syntax of formulae. In essence, we keep separated a typing context and a multiset of predicates. An invariant of the type system states that context entries do not contain refinement types at the top level. The type equivalence procedure (basically, equality of regular infinite trees) may use (hence, remove) predicates from the multiset, if required.

[1] A type is contractive if it contains no sub-expression of the form **rec** a_1 … **rec** $a_n.a_1$.

[2] To keep types manageable we did not expand polyadic message passing.

The rules for **assume** and **assert** in [1] are not algorithmic. Nevertheless, algorithmic rules are easy to obtain. Processes of the form **assume** A add A to the formulae multiset after breaking the tensors and eliminating occurrences of **unit**; processes of the form **assert** A try to remove the predicates in A from the multiset. Input processes of the form x?y.P eliminate the top-level refinements in the type for y; the resulting type is added to the typing environment, the predicates are added to the multiset. The remaining rules remain as in [25], except that they now work with the new procedure for type equivalence.

5 Conclusion and Future Work

We presented SePi, a concurrent programming language based on the monadic pi-calculus where communication between processes is governed by session types and where linearly refinement types may be used to specify properties about the values exchanged. In order to facilitate programming we added to SePi a few derived constructs, such as output and input of multiple values, mutually recursive process definitions and type declaration, session initiation, as well as the **dualof** type operator.

Our early experience with the language unveiled a few further constructs that may speed up code development, such as, a simple import clause allowing the inclusion of code in a different source file, thus providing for limited support for API development. In order to keep the language simple, the current version of SePi uses predicates over values only, thus preventing formulae containing expressions, such as p(x+1). We plan to add expressions to predicates, together with the appropriate theories (e.g., arithmetic), combining the current type checking algorithm with an SMT solver. Finally, we acknowledge that the current language of formulae is quite limited (essentially a multiset of uninterpreted formulae). We are working on a system that provides for the persistent availability of resources in a form of replicated (or exponential) resources. Polymorphism and subtyping may be incorporated in future versions of the language. We are also interested in extending the type system so that it may guarantee some form of progress for well typed processes.

Acknowledgements. This work was partially supported by project PTDC/EIA–CCO/1175 13/2010 and by LaSIGE (PEst-OE/EEI/UI0408/2011). We are grateful to Dimitris Mostrous, Hugo Vieira, and to anonymous referees for their feedback.

References

1. Baltazar, P., Mostrous, D., Vasconcelos, V.T.: Linearly refined session types. In: 2nd International Workshop on Linearity, vol. 101 of EPTCS, pp. 38–49 (2012)
2. Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C., Leifer, J.J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: Computer Security Foundations Symposium, pp. 124–140. IEEE (2009)

3. Boudol, G.: Asynchrony and the pi-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis (1992)
4. Caldeira, A., Vasconcelos, V.T.: Bica. http://gloss.di.fc.ul.pt/bica
5. Campos, J., Vasconcelos, V.T.: Channels as objects in concurrent object-oriented programming. In: 3rd International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, vol. 69 of EPTCS, pp. 12–28 (2011)
6. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. Oper. Syst. Rev. **40**(4), 177–190 (2006)
7. Gay, S., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Principles of Programming Languages, pp. 299–312. ACM (2010)
8. Gay, S.J., Hole, M.J.: Subtyping for session types in the pi calculus. Acta Inf. **42**(2/3), 191–225 (2005)
9. Girard, J.-Y.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
10. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
11. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
12. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Principles of Programming Languages, pp. 273–284. ACM (2008)
14. Hu, R., Yoshida, N., Honda, K.: Session-based distributed programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008)
15. Milner, R.: Functions as processes. J. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992)
16. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Inf. Comput. **100**, 1–77 (1992)
17. Neubauer, M., Thiemann, P.: An implementation of session types. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 56–70. Springer, Heidelberg (2004)
18. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012)
19. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Plotkin, G.D., Stirling, C.P., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner, pp. 455–494. MIT Press, Massachusetts (2000)
20. Sackman, M., Eisenbach, S.: Session types in Haskell: updating message passing for the 21st century. Technical report, Department of Computing, Imperial College (2008)(2008)
21. Turner, D.N.: The polymorphic Pi-calculus: theory and implementation. Ph.D. thesis, University of Edinburgh (1995)
22. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Pareschi, R. (ed.) ECOOP 1994. LNCS, vol. 821, pp. 202–218. Springer, Heidelberg (1994)

23. Vasconcelos, V.T.: TyCO gently. DI/FCUL TR 01–4, Faculty of Sciences, Department of Informatics, University of Lisbon (2001)
24. Vasconcelos, V.T.: Sessions, from types to programming languages. Bull. Eur. Assoc. Theor. Comput. Sci. **103**, 53–73 (2011)
25. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012)

Springer

http://www.springer.com/978-3-319-05031-7

Software Engineering and Formal Methods
SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS,
FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain,
September 23-24, 2013, Revised Selected Papers
Counsell, S.; Núñez, M. (Eds.)
2014, XXV, 432 p. 150 illus., Softcover

ISBN: 978-3-319-05031-7