Typed First-class Communication Channels and Mobility for Concurrent Scripting Languages

Paweł T. Wojciechowski

Poznań University of Technology, Poland
Pawel.T.Wojciechowski@cs.put.poznan.pl

Abstract. In the 1990s, there was considerable interest in mobile computation: systems in which running computations (or mobile agents) could be moved from one machine to another. Much of this work was in terms of high-level programming languages and mobile process calculi. An example is Nomadic Pict—a prototype high-level programming language in which to express and verify overlay networks, for reliable communication between mobile agents. One can ask whether the language abstractions could be useful for scripting programming in modern distributed deployment platforms, such as many-core processors, grids, web servers and datacentres. In this paper, we demonstrate selected features of Nomadic Pict, and show the use of typed channels and agent mobility for programming in the grid. We demonstrate example design patterns that can be used for implementing safe message passing, test & send, system bootstrapping, and relocatable computation.

1 Introduction

In the 1990s, there was considerable interest in *mobile computation*: systems in which running computations could be moved from one machine to another. Much of this work was in terms of high-level programming languages and mobile process calculi, such as the π-calculus [11] and Mobile Ambients [5] (see e.g. [14, 10, 8, 12, 19, 17] among others). Process calculi (also known as process algebras) were originally conceived for the formal study of concurrent and mobile communication systems. They provide a rigorous framework where complex systems can be accurately analyzed, including reasoning techniques to verify their essential properties. In parallel, various high-level programming languages have been designed based on the process calculi. Unfortunately a lot of this work still remains theoretical, with only a few language implementations available. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. One can ask whether the semantic theory and language abstractions developed in these frameworks could be applied (or adapted) to *scripting languages* designed for distributed deployment platforms such as many-core processors, grids, web servers and datacentres?

One of the goals of scripting languages for distributed deployment platforms is to provide a lightweight but expressive set of programming constructs for connecting distributed chunks of computations (or whole applications, services, etc.)

and defining control flow. In such languages, some programming errors can be detected via *type-checking*, either statically or, more often, dynamically. Important modern scripting languages include Perl, Python, PHP, JavaScript, Ruby or extensions of Lisp. Many of these languages were originally developed for specialized domains, e.g. web services, but are increasingly being used more broadly. A shortcoming of most scripting languages is the lack of first-class support for concurrency. Concurrency is nowadays ubiquitous and no longer bound to a narrow high-performance computing domain. It is required for scalability and interacting with remote services. The newest proposals of scripting languages overcome these shortcomings. For example, Thorn [4] has support for concurrency based on message passing between lightweight, isolated processes. Clojure [6], an extension of Lisp, takes a different approach to concurrency and supports sharing changing state between threads in a synchronous and coordinated manner using Software Transactional Memory (STM). Typed message-passing is the sole means of communication between processes in the Singularity OS [7]. However, the implementation does not support cross-machine channels.

On the other hand, relocatable computation is not yet a frequently supported feature in scripting languages. In the virtualized environments, relocatable computation (or virtual machines) can make it easier to deploy applications and reduce the impact of partial system failures by moving applications from a misbehaving network node to a non-faulty node. Thus, the design of novel scripting languages that support mobile computations could improve system robustness. Are tasks typical of scripting programming that would best be expressed at the level of mobile process calculi? In this paper, we describe a series of programming examples (or patterns) that answer positively. We target the grid platform but the patterns presented in the paper are general, and so can be applied to other deployment platforms as well. In [2], the authors describe type-safe programming mechanisms for combining and managing enterprise services, in the setting of farms of virtual machines. It would be interesting future work to extend our language to control VMs, using the service combinators described in [2].

In the late 1990s, we developed *Nomadic Pict* [18, 19, 17, 13][1]—a mobile agent distributed programming language. The low-level language extends the compiler and run-time system of Pict [14], a concurrent language based on the π-calculus, to support our primitives for agent creation, migration, and location-dependent communication. High-level languages, with particular infrastructures for location-independent communication, can then be obtained by applying user supplied translations into the low-level language. An experimental implementation of Nomadic Pict and further details are available from [13]. The goal of this paper is to show how Nomadic Pict's abstractions, such as typed first-class communication channels and agent mobility could be used to safely express typical tasks of a scripting language for distributed deployment platforms. We use concrete examples of executable programs in Nomadic Pict to express: safe message-passing communication, test & send synchronization, system bootstrapping, and relocatable computation in a networked system. The examples are toy

[1] Nomadic Pict and its theory is joint work with Peter Sewell and Asis Unyapoth.

applications that serve only to illustrate the concepts, but we hope that abstractions such as typed first-class channels and relocatable computation will pave the way into future industrial strength scripting languages.

Some of the Nomadic Pict abstractions have been encoded in libraries of general-purpose functional programming languages. For example, an experimental language Acute [15] has a distributed message-passing library that is an implementation of the Nomadic Pict constructs for migration of mobile computations and communication between them. Acute extends an ML-like core language to support distributed development, deployment, and execution, allowing type-safe interaction between separately built programs. Some of these ideas were further developed and put into practice in HashCaml language [3], an extension of the OCaml bytecode compiler with support for type-safe marshalling and related naming features.

This paper is *not* a research paper on Nomadic Pict but a paper to accompany a language demonstration. The readers interested in the design and implementation of our language, the Nomadic π-calculi, formal reasoning and proofs are referred to [18].

2 Language Demonstration

2.1 Typed channels for safe communication

Consider a program in which several parallel processes communicate by means of messages. One of the frequent programming errors in message-passing programs is that the type of values marshaled for communication does not match the type of values expected on the receiver's side. Below is a small program in Nomadic Pict to illustrate this case.

```
new x : ^Int      {- Communication channel creation -}

run (
  x! 10
| x? msg= printi! msg      {- Message-passing communication -}
)
```

The above program creates a communication channel using a keyword `new`; the channel is named `x`, and has a type `^Int` of channels carrying values of type `Int`. Then, the program executes (using `run`) two parallel processes. The first process outputs a message (a value `10`) on channel `x`, and terminates. In parallel, denoted with `|` (bar), the second process waits for an input on channel `x`. After the message has been received, it is substituted for the formal parameter `msg` and the process reduces to `printi!10`, which prints out `10`.

If the program would be modified, so that the first process outputs a message of a different type, e.g. a record of two integers `[10 10]`, or the second process expects a value of a different type than `Int`, then the program would not be correct, and the Nomadic Pict compiler would generate an error. Later in the paper, we show programs that use typed channels for network communication. In such programs, the same typing principle is used, allowing type-mismatch errors to be detected at compile time. This simple typing principle could be further extended, e.g. to support session types [9].

Processes communicate using message passing instead of shared variables, which removes the need for locks. Channel names are *first-class* values, i.e. they can be created at runtime and passed as arguments or results of function calls. Contrary to the message-passing languages that follow the Actor model (such as Erlang [1]), channel names can be passed along other channels in the style of the π-calculus. For instance, in the following program a channel name `x` will be communicated on another channel of type `^^Int` to some other process executed in parallel, which can use `x` for communication. First-class communication channels can be very useful in grid programming, e.g. to dynamically reconfigure the logical network topology of a grid in response to some events.

```
new x : ^Int       {- Creation of typed first-class channels -}
new y : ^^Int

run (
  x! 10 | y! x | y? p= p? msg= printi! msg    {- Communication -}
)
```

The program above creates two communication channels, named `x` and `y`, using a keyword `new`. The channels are typed. The former channel has type `^Int` of channels that can only carry values of type `Int`, while the latter channel has type `^^Int` (understood as `^(^Int)`) of channels that can carry names of channels of the former type. In the main part of the program we execute (using `run`) three parallel processes. The first and second process output their messages, respectively on channel `x` and `y` and terminate, while the third process waits for an input on channel `y`. The output and input on channel `y` can synchronize, reducing the third process to an input process `x?arg= printi!arg`, which again synchronizes with an output on `x`. Finally, the program prints out the message received on channel `x`, i.e. `10`.

The construct `<chan>?<pattern>=` is only used for one input. If we would require the input process to be ready to accept new messages, then we should use a replicated input construct, as in the program below.

```
new x : ^Int

run (
  x!1 | x!2 | x!3         {- Three parallel output processes -}
| x?* arg = printi! arg   {- A replicated input process (server) -}
)
```

In the above program, three concurrent processes output integer numbers, which are received by another process (a server) that prints them all out. The order of message delivery is unspecified, since the parallel processes in our program are not synchronized. In case of remote communication, we may choose to replace the `x?* arg = ...` construct by a timed input as below.

```
wait
  x?* arg = printi! arg
timeout
  t -> print! "Timeout!"
```

If no message is received on channel `x` after `t` seconds (roughly), then an exception is raised and handled in the `timeout` clause.

2.2 Agents and test & send

A *distributed computation* is one whose portions can be executed in different sites (or grid nodes, or processors) interconnected via a network. In Nomadic Pict a distributed computation consists of *agents* located on sites, where a *site* is an instance of the Nomadic Pict runtime system. Internally, agents may consist of many concurrent processes that can communicate using channels. The channels are distinct, in that outputs and inputs can only interact if they are in the same agent. This provides a limited form of *dynamic binding*, with the semantics of a channel name (i.e., the set of partners that a communication on that channel might synchronise with) dependent on the agent in which it is used.

In order to test if an agent is present on a local site, we can use a *test & send* synchronization construct `iflocal`.

```
new chan : ^String

agent a =
 iflocal <b> chan! "MESSAGE"
   then print! "b is on this site."
   else print! "b is not here."

and b =
  chan ?* msg = print! msg
```

The above program creates two agents `a` and `b`. Execution of the conditional `iflocal <b>chan!"MESSAGE"` by agent `a` checks if agent `b` is on agent `a`'s current site. If so, then it delivers a message `"MESSAGE"` to channel `chan` inside agent `b` as part of the same atomic action, and continues with the 'then' clause. Otherwise, it continues with the 'else' clause. The `iflocal` construct may simplify programming of failure detectors in the grid.

2.3 Distributed bootstrapping

Grid computations are to be executed on a large number of machines. Therefore, parallel portions of a distributed computation must be spawned on machines automatically, with any communication links properly established. If the programming environment does not offer any support of this sort, the programmer has to implement bootstrapping of a grid system. Below we demonstrate how this can be done in Nomadic Pict.

Execution of `migrate to n` migrates the whole agent including any communication channels to a site `n`. After migration, the agent's execution commences from the point in which it has stopped before migration. Migration transparency greatly simplifies programming, for the cost of a more complex virtual machine.

```
val n = ''sirius.cs.put.pl'':5000
new c : ^String

agent a =
(
  migrate to n     {- Migrate agent a to sirius and continue -}
  c ?* msg = print! msg
)

and b =
  <a @ n> c! "Hello!"
```

In the above program, two agents `a` and `b` are created. After creation the former agent migrates to a site `n`, identified by a pair of an IP address and a port number and waits for a message on channel `c`. In parallel, agent `b` outputs a string message `"Hello!"` to agent `a`, and terminates. Agent `a` is expected to be on site `n`. If the agent will not be there, when the message has arrived, the message is discarded. (Alternatively, a message could be sent to a static daemon agent that uses `iflocal` to deliver messages locally.)

The Nomadic Pict language also has a construct `<a>c!m` for *location-independent (LI)* communication, which does not require the agent's site to be specified. An application-specific *overlay network* will deliver message `m` to agent `a` irrespective of its current location. It is guaranteed that the message will be delivered despite of any agent migrations. Different LI overlay networks can be chosen from the package; the choice depends on the application.

2.4 Relocatable computation

In a grid system, it is inevitable that some machines may partially fail or slow down. Thus, the grid admins should be able to relocate processes running on these machines to non-faulty nodes. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. In a recent paper, we discuss the use of the Nomadic Pict calculus for verifying overlay networks for relocatable computations [16].

Below we demonstrate the use of relocatable computation for active messages. Let us assume that on the site `''sirius.cs.put.pl'':5000`, an agent `Smith` has been created, waiting for a message:

```
new ch : ^String

agent Smith =
  ch?* msg= print! msg
```

On another site, a function `dispatch` is defined that spawns an agent `messenger` for delivering message content of type `X`, to a recipient described using a triple of agent, site and channel names passed as arguments. After `messenger` is created, the function returns a value `0`. Below the function is called, resulting in `messenger` migrating to `Smith`'s site and delivering a message locally. The recipient's agent/channel names can be obtained using a name server (see Section 2.7).

```
def dispatch (#X  a:Agent  s:Site  c:^X  msg:X) : Int =
(  agent messenger =
   (
     migrate to s
     iflocal <a> c! msg
       then print! "OK, delivered."
       else print! "No recipient."
   )
0)

val stat = (dispatch Smith ''sirius.cs.put.pl'':5000 ch "Hello!")
```

To support different types of messages the channel `c` in `dispatch` has a polymorphic type (from Pict), which is defined by a type variable `X`. The type variable can be specialized to *any* type. It our example, it is specialized to `String`, when substituted in the function call by a channel `ch` of type `^String`.

The migrating agent `messenger` can be an arbitrary program, e.g. a presenter of the e-mail content. Thus, we can dynamically add some new computation on `Smith`'s node, even if the original program on this node was not designed for this. If needed, `messenger` could also voluntarily relocate to another server using `migrate to` and continue computation there.

2.5 Types for input/output modalities

In some programs, we may intend a communication channel to be used only for inputs or only for outputs. Otherwise, a program may be incorrect. Below we illustrate the use of the Pict type system for safe programming of input/output modalities. This type system has been extended in Nomadic Pict for distributed programming. Below is an example program implementing a server function, which creates on demand (in response to the function call) a fresh channel that can be used for communication with the server, as explained below.

```
def server () : !Int =
 (new c : ^Int
  run c? msg = printi! msg
  c)

val x = (server)
run x! 10
```

Execution of the above function `server` creates a fresh communication channel `c` carrying integers, and waits for a message on it. The returned channel is assigned to a variable `x` that is later used for an output (of a value `10`). The type of channels returned by the function is `!Int` instead of `^Int`, where `!` (exclamation mark) means that the channel name has only an *output capability*, i.e. it cannot be used for an input. Thus, we can guarantee that only the server process can read on this channel. This mechanism supports *confidentiality* since no other process can read from this channel.

2.6 Types for variant messages

It would be inconvenient to create and use a different channel for every new type of a message. How to communicate messages of different types in the same channel, and still be able to statically check if the types of marshaled/unmarshaled values are correct ? Below is an example program that uses a suitable mechanism, adopted in Nomadic Pict.

```
new c : ^[num>Int text>String]

run c ?* msg =
  switch msg of
  (
    num> v : Int -> printi! (+ v 1)
    text> s : String -> print! (+$ "message: " s)
  )

run  c! [num> 2]
run  c! [text> "foo"]
```

The above program creates a channel `c` that has a *variant type* of channels carrying either messages of type `num>Int` or `text>String`, where `num` and `text` are labels that differentiate between the types. A message received on this channel must be first resolved using a construct `switch ... of`. The construct allows messages to be matched against patterns, followed by corresponding actions.

In the above program, two types of messages are sent on channel `c`: an integer 2 and a string `"foo"`. The integer message is incremented and printed out, while the string message is first concatenated (using a function `+$`) with another string, and then printed out. The program does not compile if we would try to send on the `c` channel a value of a different type.

2.7 Types for dynamic messages

When programs are compiled separately and should connect each other, the usual approach is to publish names of channels/agents/sites at some *name server*. The address of this server is known to all processes in the grid, so that any new agent joining the system can get the public names and use them for communication. For example, in the program in Section 2.4, a process calling function `dispatch` could obtain the channel and agent names of the message recipient from a name server, using library functions `publish` and `subscribe`.

In untyped languages, the publish/subscribe code is prone to errors that can be difficult to find. In typed languages, type-checking of dynamic values is usually done entirely at runtime. However, the risk of producing erroneous code exists if the language does not force the programmer to implement exceptions.

Below is a code fragment of a new joining process. The program uses *dynamic types*, i.e. types that are not erasured by the compiler, but which accompany values at runtime. Dynamic types are erasured in Nomadic Pict explicitly, using a construct `typecase`, which requires exception code to be specified.

```
new c : ^Dyn

run c?* v =
  typecase v of
    [ a:Agent s:Site d: ^String ] -> <a@s> d! "Hello world!"
    stat : Int -> printi! stat
  else
    print! "Type not recognized!"

run c!(dynamic 3)
```

The above program creates a name server channel `c` that can be used for carrying messages of *any* type at the same time. Then, we implement a client process that expects only two types of messages to be received from channel `c`: either a triple of agent, site and channel names to be used for communication with the agent, or some integer value. If a message received does not match these types, exception code is executed (here, an error message is printed).

Contrary to statically checked variant types, described in Section 2.6, type-checking is done dynamically, when the values are resolved by `typecase`. A dynamic value can be created using construct `dynamic`, which marshals a value with a runtime representation of its type.

3 Conclusions

In the paper, we gave some taste of distributed programming in Nomadic Pict–a prototype, strongly-typed language based on the π-calculus. As other languages based on process calculi, it offers abstractions that are small and easy to learn. In the paper, we demonstrated the use of statically and dynamically typed first-class channels for safe message-passing communication. Notably, Nomadic Pict also supports relocatable computation—a rare feature that greatly simplifies system bootstrapping and enables active messages. We think that this sort of programming abstractions are a tool that would be useful for future concurrent scripting languages in modern deployment platforms, such as many-core processors, grids, web servers and datacentres. In the paper, we demonstrated the main features of Nomadic Pict, focusing on grid programming. This prototype serves as a proof-of-concept and lacks many features that are necessary for practical applications, such as integration with other languages and environments. It would be interesting future work to develop a concurrent scripting language for managing relocatable virtual machines, using the abstractions of Nomadic Pict.

References

1. Joe L. Armstrong and Robert Virding. Erlang – an experimental telephony switching language. In *Proc. XIII International Switching Symposium*, May–June 1991.
2. Karthikeyan Bhargavan, Andrew D. Gordon, and Iman Narasamdya. Service combinators for farming virtual machines. In *Proc. COORDINATION '08*, June 2008.
3. John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *Proc. 2006 ACM SIGPLAN Workshop on ML*, 2006.
4. Bard Bloom, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn–robust, concurrent, extensible scripting on the JVM. In *Proc. OOPSLA '09*, October 2009.
5. Luca Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science (TCS)*, 240(1):177–213, 2000.
6. Clojure. *Distribution files and documentation.* `http://clojure.org/`.
7. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. EuroSys '06*, April 2006.
8. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR '96*, LNCS 1119, August 1996.
9. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. ESOP '98*, LNCS 1381. Springer, March-April 1998.
10. Luís Lopes, Álvaro Figueira, Fernando Silva, and Vasco T. Vasconcelos. A concurrent programming environment with support for distributed computations and code mobility. In *Proc. CLUSTER 2000*, November 2000.
11. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
12. Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE TSE*, 24(5):315–330, 1998.
13. Nomadic Pict Language. `http://www.cs.put.poznan.pl/pawelw/npict`.
14. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
15. Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proc. ICFP '05*, Sept. 2005.
16. Peter Sewell and Paweł T. Wojciechowski. Verifying overlay networks for relocatable computations (or: Nomadic Pict, relocated). In *Proc. Workshop on the Rise and Rise of the Declarative Datacentre*, May 2008. Microsoft MSR-TR-2008-61.
17. Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31. Springer, 1999.
18. Peter Sewell, Paweł T. Wojciechowski, and Asis Unyapoth. Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM TOPLAS*, 32(4):1–63, April 2010.
19. Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April-June 2000.