The Nomadic Pict System Release 1.0-alpha

Documentation and user's manual

Paweł T. Wojciechowski

University of Cambridge
Computer Laboratory
New Museums Site
Cambridge CB2 3QG
England
`Pawel.Wojciechowski@cl.cam.ac.uk`

December 19, 2000

This draft is still being checked for consistency and completeness. Bug reports appreciated.

Contents

Copying

Nomadic Pict is available electronically from `http://www.cs.put.poznan.pl/pawelw/npict.html`

Acknowledgements

The Nomadic Pict language design is joint work with Peter Sewell and Benjamin C. Pierce. We owe special thanks to Benjamin and David N. Turner for allowing us to use the source code of Pict. We thank Robin Milner for creating a motivating environment for inventing a new language in the predominantly Java and C oriented world. Robin Milner's past and present work on ML and the π-calculus in particular is strongly in the background of this project.

Foreword

The primary goal of the Nomadic Pict project, begun at the University of Cambridge in 1996, was to design and implement a low-level language based on process calculi which offers good abstractions for distributed programming (with thread mobility) in a system where machines and communication links may crash. This report describes the syntax and use of Nomadic Pict Release 1.0 – the first implementation of Nomadic Pict, and makes little attempt to explain or motivate the Nomadic Pict language design. Interested readers are directed to the definition of the Nomadic π-calculus of Sewell, Wojciechowski, and Pierce [SWP99], and a brief comparison of Nomadic Pict with similar languages, included in [Woj00a].

The implementation of Nomadic Pict described here is built on Pict of Pierce and Tuner[PT97a, PT97b, Tur96], a concurrent (but not distributed) language based on the asynchronous π-calculus[MPW92, HT91, Bou92][1]. Pict supports fine-grain concurrency and the communication of asynchronous messages between parallel threads. We use primitives of Pict to express computation within an agent. Nomadic Pict has a two-level architecture. The Low-Level Nomadic Pict extends Pict with primitives for agent creation, migration of agents between sites, and communication of location-dependent asynchronous messages between agents. The high-level language adds location-independent communication – an arbitrary distributed infrastructure to support this communication can be expressed as a user-defined translation into the low-level language. The translation encoding defines actual properties of the system (e.g. as for robustness and tolerance to system failures). Nomadic Pict has a standardised low-level runtime system that is common to many machines, with divergent high-level facilities chosen and installed on demand. It has been implemented in O'Caml. In principle, it should run on any platform that O'Caml supports.

This document is not intended as a tutorial on distributed programming with mobility, but no previous experience in distributed programming is required. However, we do assume some familiarity with the Pict language syntax and concurrent programming in the style of the π-calculus. Readers who find it hard to understand the language notation and examples included in this document should begin with the tutorial on Pict programming[PT97c]. Below, we outline the contents of this report. After a small example illustrating the principles of distributed programming in our system, Chapter 1 describes the primitives of Low-Level Nomadic Pict. Chapter 2 then defines High-Level Nomadic Pict and the language for expressing translations from high- to low-level. Chapter 3 discusses derived language forms and useful programming idioms. Chapter 4 gives a small example application in Nomadic Pict, illustrating the expressive power of the language, and presents an example communication infrastructure encoding. Chapter 5 explains how to compile and execute Nomadic Pict programs. Chapter 6 presents the concrete syntax. Libraries are described in a separate document[Woj00b].

[1] We extended the original Pict compiler and ported many of Pict libraries. It should be possible to compile Pict-4.* programs (modulo libraries) and execute them in the Nomadic Pict runtime system. The Pict source code is available from `httl://www.cis.upenn.edu/ ~ bcpierce` and distributed under the terms of the GNU General Public License as published by the Free Software Foundation.

Chapter 1

Low-Level Language

The Nomadic Pict language has a two-level architecture. Low-Level Nomadic Pict extends Pict with primitives for agent creation, migration of agents between sites, and communication of location-dependent asynchronous messages between agents. The high-level language adds location-independent communication; an arbitrary distributed infrastructure to support this communication can be expressed as a user-defined translation into the low-level language using the modularisation facilities of the language. It is possible to deploy such infrastructure dynamically using migration.

We begin with a simple example. Below is a program in the low-level language showing how an applet server can be expressed. It can receive (on the channel named `getApplet`) requests for an applet; the requests contain a pair (bound to `a` and `s`) consisting of the name of the requesting agent and the name of its site.

```
getApplet ?* [a s] =
  agent b =
     migrate to s
     ( <a@s>ack!b  |  ... )
  in ()
```

When a request is received the server creates an applet agent with a new name bound to `b`. This agent immediately migrates to site `s`. It then sends an acknowledgement to the requesting agent `a` (which is assumed to be on site `s`) containing its name. In parallel, the body `...` of the applet commences execution.

The example illustrates the main entities of the language: sites, agents and channels. *Sites* should be thought of as physical machines or, more accurately, as instantiations of the Nomadic Pict runtime system on machines; each site has a unique name. Sites are unstructured; neither network topology nor administrative domains are represented in the language. *Agents* are units of executing code; an agent has a unique name and a body consisting of some Nomadic Pict process; at any moment it is located at a particular site. *Channels* support communication within agents, and also provide targets for inter-agent communication—an inter-agent message will be sent to a particular channel within the destination agent. Channels also have unique names. The language is built above asynchronous messaging, both within and between sites; in the current implementation inter-site messages are sent on TCP connections, created on demand, but when writing Nomadic Pict programs you should *not* depend on the message ordering that could be provided by TCP. The inter-agent message `<a@s>ack!b` is characteristic of the low-level language. It is location-dependent—if agent `a` is in fact on site `s` then the message `b` will be delivered, to channel `ack` in `a`; otherwise the message will be discarded. In the implementation at most one inter-site message is sent. Below we describe the primitives of the low-level language.

1.1 Primitives

We will introduce the low-level primitives in groups. They fall into two main syntactic categories of *processes* and *declarations*. A program is simply a series of declarations, which may contain processes. For simplicity, we confuse other syntactic categories such as abstractions, patterns, values, paths, types, and constants.

Some of them are described informally in following sections. Chapter 6 contains a complete definition of the concrete syntax in a notation similar to the Backus-Naur Form.

1.1.1 Declarations

Declarations `D` include definitions of types, channels, process abstractions, agents, and also a migration primitive.

`type T = T'`	type abbreviation
`new c:T P`	new channel name creation
`agent a=P and ... and a'=P' in Q`	agent creation
`migrate to s P`	agent migration
`def f[...]=P and ... and f'[...]=P' Q`	process abstraction

The declaration `typeT = T' P` introduces a new name `T` for complex type `T'`. Execution of `new c:^T P` creates a new unique channel name for carrying values of type `T`; `c` is binding in `P`. The execution of the construct `agent a=P in Q` spawns a new agent on the current site, with body `P`. After the creation, process `Q` commences execution, in parallel with the rest of the body of the spawning agent. The new agent has a unique name which may be referred to both in its body and in the spawning agent (i.e. `a` is binding in `P` and `Q`). A group of agent definitions introduced by `agent` and separated by `and` can be mutually recursive, i.e. each of the bodies `P` can refer to any of the defined agent names. Agents can migrate to named sites — the execution of `migrate to s P` as part of an agent results in the whole agent migrating to site `s`. After the migration, process `P` commences execution in parallel with the rest of the body of the agent. The `def` declarations are used to define process abstractions (i.e. process expressions prefixed by patterns); they are described in Chapter 3.

1.1.2 Processes

Processes `P,Q,...` form a separate syntactic category.

`(P | Q)`	parallel composition
`(D P)`	local declaration
`()`	null process

The body of an agent may consist of many process terms in parallel, i.e. essentially of many lightweight threads. They will interact only by message passing. We can write a composition of more than two processes as `(P1 | ... | Pn)`. Large programs often contain processes with long sequences of declarations like `(new x1:T1 ... (new x2:T2 P))`. We can avoid many nested parentheses and simply write `(new x1:T1 ... new x2:T2 P)`. In sequences of declarations, it is convenient to start some process running in parallel with the evaluation of the reminder of the declarations. We can use the Pict declaration keyword `run` for this purpose, e.g. a program

```
(new x:T
 run print!"Hello"
 new y:T
 P)
```

will be transformed into `(new x:T (print!"Hello" | (new y:T P)))`

`c!v`	output `v` on channel `c` in the current agent
`c?p = P`	input from channel `c`
`c?*p = P`	replicated input from channel `c`
`if v then P else Q`	conditional

To express computation within an agent, while keeping a lightweight implementation and semantics, we include π-calculus-style interaction primitives of Pict. An output `c!v` (of value `v` on channel `c`) and an input c?p=P in the same agent may match, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. The communication is asynchronous, i.e. the output is never blocked. The implementation uses local environments to store bindings of parameters to actual values. A replicated input c?*p=P behaves similarly except that it persists after the matching, and so may receive another value. In both c?p=P and c?*p=P the names in p are binding in P.

iflocal <a>c!v then P else Q	test-and-send to agent a on this site
<a>c!v	send to agent a on this site
<a@s>c!v	send to agent a on site s
wait c?p=P timeout t -> Q	input with timeout

Finally, the low-level language includes primitives for interaction between agents. The execution of iflocal <a>c!v then P else Q in the body of an agent b has two possible outcomes. If agent a is on the same site as b, then the message c!v will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will be discarded, and Q will execute as part of b. The construct is analogous to test-and-set operations in shared memory systems — delivering the message and starting P, or discarding it and starting Q, atomically. The test-and-send can greatly simplify algorithms that involve communication with agents that may migrate away at any time, yet is still implemented locally, by the runtime system on each site. Two other useful constructs can be expressed in the language introduced so far (using migration, agent creation, and test-and-send): <a>c!v and <a@s>c!v attempt to deliver c!v to agent a, on the current site and on s respectively. They fail silently if a is not where expected and so are usually used only where a is predictable. For implementing infrastructures that are robust under some level of failure, or support disconnected operation, some timed primitive is required. The low-level language includes a single timed input as above, with timeout value n. If a message on channel c is received within n seconds then P will be started as in a normal input, otherwise Q will be. The timing is approximate, as the runtime system may introduce some delays.

terminate	terminate execution of the current agent

We also include constructs for garbage collection. The execution of terminate terminates the current agent and removes its closure from the heap, releasing memory occupied by the agent. Any I/O operations (e.g. input from a keyboard) will be abruptly killed.

1.2 Names and Scope Extrusion

Names play a key rôle in the Nomadic Pict language. New names of agents and channels can be created dynamically. These names are *pure*, in the sense of Needham [Nee89]; no information about their creation is visible within the language (in our current implementation they do contain site IDs, but could equally well be implemented by choosing large random numbers). Site names, contain an address and port number of the runtime system which they represent.

Channel, agent, and site names are first-class values and they can be freely sent to processes which are located at other agents. As in the π-calculus, names can be *scope-extruded* — here channel and agent names can be sent outside the agent in which they were created. For example, if the body of agent a is

```
agent b =
(
  new d : T
  iflocal <a>c!d then () else ()
)
in
  c?x=x![]
```

then channel name d is created in agent b. After the output message c!d has been sent from b to a (iflocal) and has interacted with the input c?x=x![] there will be an output d![] in agent a.

We require a clear relationship between the semantics of the low-level language and the inter-machine messages that are sent in the implementation. To achieve this we allow direct communication between outputs and inputs on a channel only if they are *in the same agent* — messages can be sent from one agent to another only by `iflocal` (and derived forms for sending to an agent on a site, such as `<a>c!v` and `<a@s>c!v`). Intuitively, there is a distinct π-calculus-style channel for each channel name in every agent. For example, if the body of agent `a` is

```
agent b =
(
  new d : T
  (d?=()
  | iflocal <a>c!d then () else ())
)
in
  c?x=x![]
```

then after some reduction steps `a` contains an output on `d` and `b` contains an input on `d`, but these cannot react. At first sight this semantics may seem counter-intuitive, but it reconciles the conflicting requirements of expressiveness and simplicity of the language. An implementation creates the mailbox datastructure — a queue of pending outputs or inputs — required to implement a channel as required; it is garbage collected when empty. The queue is part of an agent's state which is transferred with every move of the agent. We could further develop our example and send name `d` back to agent `b` and use it for communication with the input on `d` inside agent `b`. The output on `d` can be placed anywhere inside agent `b` (in particular outside the lexical scope of `d`) but it may still interact with the input on `d` as long as both the input and output are in the same agent.

1.3 Types

All bound variables (and wildcards) are explicitly typed. In practice, however, many of these type annotations can be inferred automatically by the compiler. Therefore we do not include them in the syntax above. Types are required in definitions, e.g. execution of `new c:^T P` creates a new unique channel name for carrying values of type `T`. The language inherits a rich type system from Pict, including simple record types, higher-order polymorphism, simple recursive types and subtyping. It has a partial type inference algorithm. Below, we summarise the key types, see [PT97b] for details.

1.3.1 Base types

The base types include `String` of strings, `Char` of characters, `Int` of integers, and `Bool` of Booleans. They are two predefined Boolean constants `false` and `true` of type `Bool`. Nomadic Pict adds new base types `Site` and `Agent` of site and agent names.

1.3.2 Channel types and subtyping

Pict's four channel types are as follows:

- `^T` is the type of input/output channels carrying values of type `T`
- `!T` is the type of output channels accepting `T`
- `?T` is the type of input channels yielding `T`
- `/T` is the type of *responsive* output channels carrying `T`; we use this type to define process abstractions and functions.

The type `^T` is a subtype of both `!T` and `?T`. That is, a channel that can be used for both input and output may be used in a context where just one capability is needed. The type `/T` is a subtype of `!T` and it was introduced to define process abstractions and channels carrying results in a functional style (see examples in Chapter 3). It should not be used for channels which are for communication between agents; we have types `^T`, `!T`, and `?T` for this. In principle, type `/T` guarantees that there is exactly one (local) receiver. We define a type `Sig` as `/[]` to mark responsive channels which are used to signal completion rather than for exchanging data.

1.3.3 Records, polymorphic and recursive types

We can use tuples `[T1...Tn]` of types `T1...Tn` and existential polymorphic types such as `[#X T1...Tn]` in which the type variable `X` may occur in the field types `T1...Tn`. We can add labels to tuples obtaining records such as `[label1=T1...labeln=Tn]`. Recursive types are constructed as `(rec X=T)`, in which the type variable `X` occurs in type `T`.

1.3.4 Variant and dynamic types

In Nomadic Pict we added a *variant* type `[label1>T1...labeln>Tn]` and a type `Dyn` of *dynamic* values. The variant type `[label1>T1...labeln>Tn]` denotes all values `[label>v:T]` such as $(label, T) \in \{(label_1, T_1), ..., (label_n, T_n)\}$, and can be used for expressing variants and types of channels carrying values of a finite set of types. The dynamic type is useful for implementing traders, i.e. maps from string keywords (or textual descriptions) to dynamic values. Dynamic values are implemented as pairs (v, T) of a value and its type.

1.3.5 Defining types and type operators

We can use a declaration keyword `type` to define new types and type operators, e.g. `type (Double X) = [X X]` denotes a new type operator `Double` which can be used as in `new c:^(Double Int)`. In Nomadic Pict programs, we often use a type operator `Map` from the libraries, taking two types and giving the type of maps, or lookup tables, from one to the other (examples of using maps are in Chapter 4).

1.4 Values and Patterns

Values Channels allow the communication of first-order values, consisting of channel, agent, and site names, constants, integers, strings, characters, tuples `[v1...vn]` of the n values `v1...vn`, packages of existential types `[#T v1...vn]`, elements of variant types `[label>v]`, and dynamic values. A dynamic value can be constructed using a constructor `dynamic`, as in `(dynamic T)`. Values are deconstructed by pattern matching on input or, in the case of variants and dynamic values, using syntactic sugar `switch` and `typecase`.

The language does not support communication of processes (except for the migration of whole agents) but for experimental reasons it permits *higher-order* functions to be communicated between agents. They will be described in Chapter 3, together with derived forms.

Characters constants are written by enclosing a single character in single-quotes, as in `'a'`. Similarly, string constants are written by enclosing a sequence of zero or more characters in double-quotes. The following escape sequences from Pict can be used to write special characters in constants:

`'`	single quote
`"`	double quote
`\`	backslash
`\n`	newline (ascii 13)
`\t`	tab (ascii 8)

Patterns `p` are of the same shapes as values (but names cannot be repeated), with the addition of a wildcard. The wildcard pattern _ instead of named variables can be used to reduce the number of irrelevant variable bindings.

Chapter 2

High-Level Language

Nomadic Pict promotes a hierarchical or layered process of building distributed applications, where each layer represents different concerns. The communication infrastructure encoding is the layer which formally defines properties of the agent communication. It can often be useful to design the encoding with an application in mind, i.e. take into account the application demands and important properties, and design a good infrastructure which gurantees these properties. This allows for better use of system resources, e.g. in wide-area and mobile networks, and when some higher levels of security are required. A more traditional approach aims at building "middleware" systems as black boxes, trying to satisfy all possible demands of all current and future applications. Below we describe the primitives of the high-level language and the language for expressing the infrastructure encodings.

2.1 Primitives

The high-level language is obtained by extending the low-level language with a single location-independent communication primitive.

`c@a!v`	location-independent output to channel `c` at agent `a`

The intended semantics of an output `c@a!v` is that its execution will reliably deliver the message `c!v` to agent `a`, irrespective of the current site of `a` and of any migrations. The low-level communication primitives are also available, for interacting with application agents whose locations are predictable. The actual semantics of `c@a!v` will depend on the encoding (or compilation-time library) of this primitive in the low-level language. The Nomadic Pict distribution files contain some example encodings which guarantee the intended semantics if the underlying distributed system is mostly reliable. Infrastructures which can tolerate different failures may be added in future.

Other low- and high-level communication primitives may be added in future, e.g. in order to support stream communication. They can be encoded as functions or using the syntax as below.

`do "key" v in P`	a placeholder for macro definition "key"

The compiler will replace each occurrence of `do "key" v in P` by a macro definition in the low-level language which has a string name `"key"`. The parameter `v` should have a type which is expected by a macro definition. Alternatively, we can simply define a function (or process abstraction) in the high-level program and reimplement this function in Low-Level Nomadic Pict using the construct {`def f ...` }`e = P`, described in §2.2. The Nomadic Pict compiler will replace function definitions in the high-level program by their equivalents defined in the compositional translation.

2.2 Expressing Encodings

The language for expressing encodings of high-level language primitives allows the translation of each interesting phrase (all those involving agents or communication) to be specified and type checked; the translation of a whole program (including the translation of types) can be expressed using this compositional translation. Below we describe the concrete syntax of the language for expressing encodings; the example infrastructure in Chapter 4 should give the idea how to use our language (see also Appendix). We will introduce the main language primitives in groups.

`program par:T = P`	Program declaration
`{toplevel P par}T' = Q`	Top level creation

The construct `program par:T = P` declares a program with body `P`, expressed in the high-level language. The name `par` of type `T` is the program parameter which may be referred to in the body `P` (i.e. `par` is binding in `P`). In order to execute a program `P` with the formal parameter `par` we need to define a toplevel using the construct `{toplevel P par}T' = Q`.

The execution of `{toplevel P par}T' = Q` spawns a new toplevel on the current site, with body `Q` expressed in the low-level language. After the creation, the runtime system commences execution of process `Q`. The names `P` and `par` are binding in `Q` and denote a user-defined program, declared by using the construct `program`, and the program parameter, which must be initiated in the toplevel body `Q`. The type `T'` in `{toplevel P par}T' = Q` is the type of the translation parameter (explained below), not to be confused with the type `T` of the program parameter `par` in `program`.

`{P}e`	a placeholder for translation of process `P`

We can use a placeholder `{P}e` inside the toplevel body `Q`. The Nomadic Pict compiler will replace `{P}e` by translation of program `P` into the low-level language, i.e. program `P` with all high-level language primitives replaced by their encodings into the low-level language. The translation has a parameter `e` of type `T'` (i.e. the type defined in the `toplevel` phrase). The parameter value must be initialised in the body `Q`.

`{Agent} = T`	translation of type `Agent`
`{Site} = T`	translation of type `Site`

In the infrastructure encoding, we may want to store additional data in values of type `Agent` and `Site`, such the name of a daemon agent on a site, or the address of a local server. This will require to encode types `Agent` and `Site` as tuples of basic types. The constructs `{Agent} = T` and `{Site} = T` are used to define the translation of types `Agent` and `Site` into complex types. The compiler will use this translation to type check the compositional translation of the high-level language primitives into the low-level language.

{agent a = P in Q}e = Proc	translation of agent creation
{migrate to s P}e = Proc	translation of migration
{c@a!v}e = Proc	translation of location-independent output
{c?*p=Q}e = Proc	translation of replicated input
{<a@s>c!v}e = Proc	translation of output to agent on site
{<a>c!v}e = Proc	translation of output to adjacent agent
{iflocal <a>c!v then P else Q}e = Proc	translation of test-and-send to agent
{wait c?p=P timeout t->Q}e = Proc	translation of timed input

These are constructs of the compositional translation of the high-level language into the low-level language. The first three constructs are the most often used. We can usually omit the rest since the translation is trivial. `Proc` in each clause of the compositional translation is the process in the low-level language which will replace the primitive of the high-level language. The compositional translation of each high-level language phrase has a translation parameter `e` whose value must be initialised in the toplevel and passed to the encoding by using the construct `{P}e` (described above).

`!a`	a pattern refering to translation variable `a`

Names `a`, `par`, and `p` in translation definitions: `{agent a = P in Q}e = Proc`, `{toplevel P par}T = Proc` and `{c?*p=Q}e = Proc`, are binders which should be created inside `Proc`. We can use pattern `!a` in `Proc` in order to refer to translation variable `a` and, e.g. assign value `v` to `a` using `val !a=v`.

`{do "key" x in P }e = Proc`	macro definition "key"

A rudimentary module system allows encodings of any new phrases of the high-level language to be expressed as macro definitions. We can use the macros in programs writing `do "key" x in P`. Here, the type of `x` is not known until the macro definition is applied and the type information can be inferred.

`{def f}e [...]  = Proc`	Redefinition of process abstraction
`{def f}e (...):T = Proc`	Redefinition of function

All process abstractions and functions in the high-level language which have types (of parameters or returned values) containing `Site` or `Agent` which have been encoded as complex types must be reimplemented in the low-level language. To express these translations we use the constructs `{def f}e [..]= P` and `{def f}e (..):T = P`. The Nomadic Pict compiler will use a new definition `P` to generate the executable code for `f`, thus replacing any implementations of `f` which were in the original program.

Chapter 3

Derived Forms and Idioms

Below we give some useful syntactic sugar and programming idioms. Most are standard distributed programming idioms such as remote procedure calls (RPC) and distributed objects; other idioms are more experimental such as composite events.

3.1 Syntactic Sugar

The core language described in Chapters 1 and 2 lacks some constructs which are useful in programming. In order to avoid complicating the semantics of the core language, additional programming features are made as *syntactic sugar*, i.e. there is an unambiguous translation of the code with the additions into code without them. Below we describe some syntactic sugar. Most are standard Pict forms; some are new. Interested readers are directed to a formal description of the source-to-source translations in Pict in [PT97b], where all Pict forms are described in detail.

3.1.1 Process abstractions and a functional style

In Pict, we can define *process abstractions*, i.e. process expressions prefixed by patterns, via the declaration keyword `def`, as in

```
def f [x:T1 y:T2] = (x!y | x!y)
```

and instances are created using the same syntax as output expressions, as in `f![a b]`. The name `f` has type `/[T1 T2]`. Recursive and mutually recursive definitions

```
def f [..] =  ...  g![..]  ...
and g [..] =  ...  f![..]  ...
```

are also allowed.

A *functional style* is supported by a small extension to the syntactic class of abstractions. For example, we can replace a process abstraction `def f [a1:T1 a2:T2 r:/T] = r!v`, where `v` is some complex value, by a 'function definition'

```
def f (a1:T1 a2:T2) : T = v
```

and avoid explicitly giving a name to the result channel `r`. For simplicity, we often confuse process abstractions as above and process abstractions which do not return any values, using a single term "functions".

We can define anonymous abstractions as in Pict

```
\[...] = ...
```

For example, below is a function `f` which accepts process abstractions of type `String -> Sig`

```
def f g:/[String Sig] = ((g "foo"); ())
```

We can create an instance of `f` passing an anonymous function which prints an argument `s` and sends an acknowledment signal on channel `r` as follows

```
f!\[s:String r:Sig] = ((pr s); r![])
```

Functions can be effectively used in Nomadic Pict by *all* agents which have been defined in the lexical scope of function definitions. So formally it looks as though each agent has a private copy of each function it might ever use. Similarly, any public library functions can be used in all agents defined in the program which has imported these libraries. Declarations of library modules precede lexically any program declarations, therefore the names of library functions are visible inside any agent in a normal way, just as any other names defined in the lexical scope. All functions defined inside an agent are private to this agent.

3.1.2 Declaration values and applications

The syntactic category of values is extended with *declaration values* of the form `(D v)`, as in

```
c!(new d:T d)
```

The complex value is *always* evaluated to yield a simple value, which is substituted for the complex expression; the process above creates a fresh channel `d` and sends it off along `c`, as in `(new d:T c!d)`.

In value expressions, we allow the *application* syntax `(v v1 ... v2)`. For example, we can define a process abstraction

```
def double [i:Int r:/Int] = +![i i r]
```

and then, in the scope of the declaration, write `(double i)` as a value, dropping the explicit result channel `r`, e.g. `printi!(double 2)` would compute `4` and print it out on the console, using the library channel `printi`.

3.1.3 Value declarations

A declaration

```
val p=v
```

evaluates a complex value `v` and names its result. Formally, a `val` declaration `(val p=v e)` is translated using the continuation-passing translation, so that the body `e` appears inside an input prefix on the continuation channel which is used to communicate a simple value evaluated from the complex value `v`. This means that `val` declarations are *blocking*: the body `e` cannot proceed until the bindings introduced by the `val` have actually been established.

3.1.4 Other syntactic sugar

The idiom "invoke an operation, wait for a signal (i.e. a null value `[]`) as a result, and continue" appears frequently, so it is convenient to introduce `;` (semi-colon), as in

```
(v1 ...); (v2 ...)
```

for the sequence of operations `v1` and `v2`.

3.1.5 Matching variants and dynamic values

In Nomadic Pict programs we use a variant type `[label1> T1 ... labeln> Tn]` so often, that it is convenient to introduce a new construct `switch`, as in

```
c?v= switch v of
      (
        label1> p1 -> P1
         ...
        labeln> pn -> Pn
      )
```

that matches a value `v` of variant type with all the variants, chooses the one which has the same label as `v`, and proceeds with a process `P` of the matched variant.

We can compare dynamic values at runtime using the construct `typecase`, as in

```
c?v= typecase v of
       s:String -> print!s
       [s:String d:^String] -> d!s
     else print!"Type not recognised."
```

where `c` has type `^Dyn`. Instances of dynamic values are created using `(dynamic v)`. For example, `c!(dynamic ["ala" x])` in parallel with the process term above may synchronise, resulting in `"ala"` being sent along the channel `x`, `c!(dynamic "ala")` would print `"ala"`, but any other (dynamic) value sent on `c` would print an error message. The constructs `switch` and `typecase` are desugared using the value equality testing primitive. In the examples above, `switch` and `typecase` are process terms but we can also use these constructs in expressions yielding a value.

3.2 Procedures

Within a single agent one can express 'procedures' in Nomadic Pict as simple replicated inputs. Replicated inputs are useful to express server agents. Below is a first attempt at a pair-server, that receives values `x` on channel `pair` and returns two copies of `x` on channel `result`, together with a single invocation of the server.

```
new pair   : ^T
new result : ^[T T]
( pair?*x = result![x x]
| pair!v
| result?z = ... z ... )
```

This pair-server can only be invoked sequentially—there is no association between multiple requests and their corresponding results. A better idiom is below, in which new result channels are used for each invocation. The pair-server has a polymorphic type (`X` is a type variable), instantiated to `Int` by a client process.

```
type (Res X) = ^[X X]
new pair : ^[#X X (Res X)]
( pair?*[#X x r] = r![x x]
| (new result:(Res Int)  (pair![1 result] | result?z =... z ...))
| (new result:(Res Int)  (pair![2 result] | result?z =... z ...)))
```

The example can easily be lifted to remote procedure calls between agents. We show two versions, firstly for location-dependent RPC between static agents and secondly for location-independent RPC between agents that may be migrating. In the first, the server becomes

```
new pair : ^[#X X (Res X) Agent Site]
    pair?*[#X x r b s] = <b @ s>r![x x]
```

which returns the result using location-dependent communication to the agent `b` on site `s` received in the request. If the server is part of agent `a1` on site `s1` it would be invoked from agent `a2` on site `s2` by

```
new result : (Res Int)
  ( <a1 @ s1>pair![7 result a2 s2]
  | result?z = ...z... )
```

If agents `a1` or `a2` can migrate this can fail. A more robust idiom is easily expressible in the high-level language—the server becomes

```
new pair : ^[#X X (Res X) Agent]
    pair?*[#X x r b] = r@b![x x]
```

which returns the result using location-independent communication to the agent `b`. If the server is part of agent `a1` it would be invoked from agent `a2` by

```
new result : (Res Int)
  ( pair@a1![3 result a2]
  | result?z= ...z... )
```

3.3 Mobile Agents

Nomadic Pict agents are located at sites and they can freely migrate to other named sites. Agents carry their computation state with themselves and their execution is resumed on a new site from the point where they stopped on previous site. Mobile agents can exchange messages on channels. A channel name can be created dynamically and sent to other agents which can use it for communication.

Below is a program in the high-level language showing how a mobile agent can be expressed. It defines a function `spawn` (which is assumed to be part of an agent `a`), containing a definition of agent `b`. The function is invoked twice, each time creating a new agent `b`

```
new answer : ^String
def spawn [s:Site prompt:String] =
  (agent b =
     (migrate to s
      answer@a!(sys.read prompt))
   in
     ())
( spawn ! [s1 "How are you?" ]
| spawn ! [s2 "When does the meeting start?" ]
| answer ?* s = print!s
...
```

which migrates to site `s`, passed as the parameter of the function `spawn`. After migration, agent `b` prints a string `prompt` and reads from a standard input. The input read on site `s` is sent back to agent `a` on `answer` (using location-independent output) and printed out.

The location-independent message delivery (including any internal encodings of agent and site names for different addressing schemas) is not part of the runtime system — it has to be encoded explicitly in the language, using the language for expressing encodings described in Chapter 2. In Chapter 4, we give a complete example program to illustrate the idea.

3.4 Locks, Methods and Distributed Objects

The language inherits a common idiom for expressing concurrent objects from Pict [PT95]. The process

```
new lock:^StateType
  ( lock!initialState
  | method1?*arg = (lock?state = ...  lock!state'  ...)
  ...
  | methodn?*arg = (lock?state = ...  lock!state'' ...))
```

is analogous to an object with methods `method1`...`methodn` and a state of type `StateType`. Mutual exclusion between the bodies of the methods is enforced by keeping the state as an output on a lock channel; the lock is free if there is an output and taken otherwise. For more detailed discussion of object representations in process calculi, the reader is referred to [PT95]. It contains an example program illustrating how a simple *reference cell* abstraction can be defined in Pict. Below we rewrite the example to show how *distributed objects* can be expressed in Nomadic Pict. The program uses mobile agents and many of the derived forms described in previous sections.

A reference cell can be modeled by an agent with two procedures connecting it to the outside world – one for receiving `set` requests and one for receiving `get` requests. Below our cell holds an integer value (in channel `contents`) that initially contains `0`.

```
type RefInt =
[
  set=/[Agent Int Sig]
  get=/[Agent /Int]
]
```

```
def refInt [s:Site r:/RefInt] =
(
 new set:^[Agent Int Sig]
 new get:^[Agent !Int]

 agent refIntAg =
 (
   new contents:^Int
   run contents!0
   migrate to s
   ( set?*[a:Agent v:Int c:Sig]= contents?_ = (contents!v | c![])
   | get?*[a:Agent res:!Int]= contents?v = (contents!v | res@a!v))
 )
 r![
   set = \[a:Agent v:Int c:Sig] = set@refIntAg![a v c]
   get = \[a:Agent res:!Int]    = get@refIntAg![a res]
 ])
```

A function `refInt` defines two method channels `set` and `get` and creates a cell agent `refIntAg` which immediately migrates to site `s`. The cell agent maintains the invariant that, at any given moment, there is at most one process ready to send on `contents` and when methods `set` and `get` are not active, there is exatly one value in `contents`. The function `refInt` returns a record which defines an interface to procedures of the cell agent. The record contains two labelled fields with anonymous functions implementing the location-independent access to the procedures. Now, we can create two instances (objects) `cell1` and `cell2` of our cell, one on site `s1` and second on site `s2`

```
val cell1 = (refInt s1)
val cell2 = (refInt s2)

agent a =
(
  (cell2.set ag 5);
  (prNL (int.toString (cell1.get a)));
  (prNL (int.toString (cell2.get a)));
  ()
)
```

and use them in some agent `a`. The agent `a` first stores `5` at object `cell2`, then gets stored values from both objects and prints them out. Distributed objects are used in some Nomadic Pict libraries.

3.5 Higher-Order Functions

The core Nomadic Pict language does not support communication of active processes (except for the migration of whole agents). The question can be risen why not to treat process terms as values which could be communicated between agents, similarly to the class serialisation mechanism in Java? This would be a fairly serious step since we had to define the execution of the process term in all possible contexts. In fact, we have already taken a similar but moderate step allowing library and other function names defined in one place to be used by all (potentially mobile) agents which have been created in the lexical scope of the function definition. For experimental reasons, we allow functions defined by `def` to be *first-class* values, which means that any private or public functions can be sent to other agents, which can receive them on channels and invoke in the same way as their own functions. However, we make no attempt to define the semantics of higher-order functions formally. In particular, they are not part of the Nomadic π-calculus definition which is described in [SWP99].

Below is a program which defines a channel carrying functions (of type `String -> String`) and two agents `a` and `b`:

```
new c: ^/[String /String]
```

```
agent a =
(
  def f (s:String) : String = (+$ s s)
  <b@glia>c!f
)

and b =
(
    ...
  c?f= (val w = (f "Voo")
        ())
)
```

The agent `a` creates a private function `f` which returns a string given as the function parameter, concatenated with itself, and sends off this function to agent `b` which is supposed to be on site `glia`. The agent `b` receives the function on channel `c`, and invokes it, passing `"Voo"` as a parameter. The function prints `VooVoo` on a local console.

In principle, in our statically-typed language the execution of functions which have been exported to some other agent, likely to be executed on a remote machine, should not cause any runtime errors. Currently, the function code is sent together with *all* other functions and global values which will be needed by the exported function. Therefore the runtime errors "variable not bound" should not happen. Note, that if the exported function had used standard Pict channels for communication with any external processes then the function behaviour would depend on the new local context.

Higher-order functions add more expressiveness to the language but they complicate the language semantics and may prevent from some highly optimised implementations in the future. Therefore, please keep in mind that future versions may not support higher-order functions.

3.6 Distributed Composite Events

Nomadic Pict has a library (written in Nomadic Pict) which implements distributed events and a composite event language for manipulating composite event expressions (such as a sequence of event types).

A *distributed event* in Nomadic Pict is an asynchronous time-stamped message. The timestamp is the event creation time (e.g. a local clock value assigned at the event source). Events can be generated at arbitrary many distributed event sources and multicast to agents (clients) that subscribed for the event type. Events are delivered to each client according to the timestamp order. Events are distributed through a third party called an *event mediator*, which can be spawned at any location whatsoever, e.g. at a site of the event source, using a function `engine`.

Events are sent on *event channels*. An event channel is simply a standard channel created by `new`, carrying values of abstract type `Event X`, e.g.

```
new badge : ^(Event ActiveBadge)
```

The `Event` type includes an event timestamp, a sequence number, and a value of the event type. Event channels must be either created in the lexical scope of the event sources and clients, or extruded to the sources and/or clients along other channels.

Before event sources can send any values to the event channel (using `submit`) the channel has to be first declared (using `declare`) at some mediator agent which is selected for dealing with the event type (there can be many mediators in a distributed system). Before receiving events clients must register for the event type at the mediator agent (using `register`), specifying the name of the event channel. Then they can receive events by simply listening to the event channel, e.g.

```
badge ?* e = ...
```

The input is blocking, waiting for events to come.

The composite event language can be used to create *composite event channels* from standard event channels and other composite event channels. Once created, they can be used by the event client for receiving notifications about the composite events of some complex event type (there is no need for an explicit registration at mediators in this case since the library does it automatically for the client which created the channel). The composite event language (derived from the composite event language of the Cambridge Event Architecture[BBHM95, Hay96]) implements following primitives:

```
inclusiveor
without
followbywithout
followby
```

and timed versions of the above (i.e. event channels will be active only for a prescribed time). The functions `followbywithout` and `followby` are available each in four different variants to accommodate different semantics. The underscore (`_`) before and/or after the core name of the function denotes a different semantics of the primitive, e.g. if we have a sequence of events: $A1$ $A2$ $B1$ $B2$ of event types `A` and `B`, then

`A followby B`	signals an event $(A2, B1)$ only
`A _followby B`	signals $(A1, B1)$ and $(A2, B1)$
`A followby_ B`	signals $(A2, B1)$ and $(A2, B2)$
`A _followby_ B`	signals $(A1, B1)$ and $(A2, B1)$ and $(A2, B1)$ and $(A2, B2)$

For more details, see the library description in [Woj00b] and an example program in the distribution.

The mediator can (optionally) delay sending of event notifications to clients which subscribed for the event. The delay time (given as a parameter when spawning the mediator) can be used for making sure that all events of some complex composite event type, which have been created at different distributed sources at roughly the same time, will arrive to the mediator *before* the composite event is created, so that it can be created having the best knowledge possible. Of course, this works well only if the anticipated delay is set up correctly, which may not always be possible.

The event types are all basic and complex values which can be built and sent along channels (this version of Nomadic Pict allows higher-order functions, so an event type can also be a program).

Chapter 4

Example Program

In this section we describe a small application in order to illustrate Nomadic Pict features, such as mobility, communication primitives, the use of channel names outside their declaration scope, and expressing the encoding of the high- into the low-level language. The program uses libraries: `Map` for expressing maps or dictionaries, and `Graphics` for a simple graphics (based on X11). We included a complete source code of the application and infrastructure encoding. The source files can be found in the `examples` directory (see also other examples and demos).

We consider the support of collaborations within (say) a large computer science department, spread over several buildings. Most individuals will be involved in a few collaborations, each of 2–10 people. Individuals move frequently between offices, labs and public spaces; impromptu working meetings may develop anywhere. Individuals would therefore like to be able to summon their working state (which may be complex, consisting of editors, file browsers, tests-in-progress etc.) to any machine. These summonings should preserve any communications that they are engaged in, for example audio/video links with other members of the project.

To achieve this, the user's working state can be encapsulated in a mobile agent, an electronic *personal assistant (PA)*, that can migrate on demand. Below, we describe a prototype of the application, where the PA agent is used only for delivering messages to moving people. In Section 4.2 we give the encoding of an example communication infrastructure.

4.1 High-Level Architecture

In the beginning of the program we need to import libraries which are not imported by default, using a keyword `import`.

```
import "Nstd/Map"
import "Graphics/Graphics"
```

After imports we can define any global names, values, and functions defined using `def` which will be used in the main program *and* infrastructure encoding (here we do not have any such data). Then, we declare the PA program

```
program hosts : [Site Site Site] =
(
  ...
)
```

The program accepts a list of site names. Below we describe the body of the program in detail. We implement the PA application with three classes of agents: the PAs themselves, which migrate from site to site; *summoner* agents, which are static (one per site) and are used to call the PAs; and a single *name server* agent, also static, which maintains a lookup table from the textual keys of PAs to their internal agent names. They interact using location-independent communication on channel names

```
new regist   : ^[String Agent]
new summon   : ^[String Site Agent]
```

```
new name     : ^[Bool String Agent]
new mid      : ^[String Agent]
new move     : ^[Site Agent]
new notFound : ^String
new done     : ^[]
new main     : ^[]
```

The program involves finite maps from Nomadic Pict standard library. We make use of the following constructs:

```
c!(map.make eq)
```

outputs the empty map on channel `c` (where `eq` is a comparing function over the keys),

```
map.add m key v
```

returns a map containing the same binding as `m`, plus a binding of `key` to `v`; if `key` was already bound in `m`, its previous binding disappears,

```
switch (map.lookup m key) of
(
  Found> w:T -> P
  NotFound> _:[] -> Q
)
```

looks up `key` in map `m`. The `Map` library contains four additional functions: for removal, testing, and iterations. The name server below maintains a map from strings to agent names; it receives new mappings on `regist`. The map is stored as an output on the internal channel `names`. Summon requests are received on `summon`, containing a textual key and the name/site of the summoner. If the key has been registered the name server sends a migration command to the corresponding PA agent, otherwise it sends the `notFound` message to the summoner. Inquiries about agent names are received on `name`, containing the key and caller name. Replies are sent back on `name`, containing `true` and key/name if the name is found, and `false` otherwise.

```
agent NameServer =
 (new names : ^(Map String Agent)
  def eq (a:String b:String) : Bool = (==$ a b)
  ( names!(map.make eq)
  | regist?*[key PA] = names?m = names!(map.add m key PA)
  | summon?*[key s Su] = names?m =
      switch (map.lookup m key) of (
        Found> PA:Agent -> (move@PA![s Su] | names!m)
        NotFound> _:[] -> (notFound@Su!key | names!m))
  | name?*[_ key a] = names?m=
      switch (map.lookup m key) of (
        Found> PA:Agent -> (name@a![true key PA] | names!m)
        NotFound> _:[] ->  (name@a![false key a] | names!m))
  ))
```

The summoner at site `s` is as below. It displays a little window on `workstation` and waits until one of the events specified in the given `Xevent` list occurs. Then, it returns the status `stat` of the mouse and keyboard at that time and executes the function `main`. If a mouse button has been pressed it gets strings from the local console, sending them as requests to the name server. In parallel, it closes the window and after receiving a mesage on `done`, the function is repeated. The event list `Xevents` is defined using the `List` library (see [Woj00b] for details).

```
val Xevents = (cons #Event [Poll>[]]
               (list.make #Event 1 [Button_down>[]]))

agent Summoner =
(
  def err s:String = print!(+$ "Error:" s)
```

```
  def open_window () : Status =
  (
    (open_graph (+$ workstation " 100x50+0-0") err);
    (moveto 30 30); (set_color red); (draw_string "DAEMON");
    (moveto 10 15); (set_color black); (draw_string "Click mouse");
    (wait_next_event Xevents)
  )
  def main stat:Status =
    if stat.button then
      (run print!"A mouse button has been pressed"
       val key = (sys.read "Summon PA : ")
       (summon@NameServer![key s Summoner]
       | ((close_graph); done?_ = main!(open_window))
       | notFound?key= (print!(+$ key " not found!") | done![])))
    else main!(wait_next_event Xevents)
  main!(open_window)
)
```

A sample PA (identified by a string `a`) is below. It has 4 parallel components; a registration message, a loop `main` for sending messages to another PA, a replicated input that receives data from other PAs and prints it, and a replicated input that receives migration commands and executes them.

```
agent PA =
    ( regist@NameServer![a PA]
    | main?*_ =
        (name@NameServer![true (sys.read "Send to? ") PA]
        | name?[ok key dest]=
           if ok then
             mid@dest![(sys.read (+$ "Type to " key)) PA]
           else (print!(+$ key " not found.") | main![]))
    | mid?*[d source] = (print!(+$ "Incoming: " d) | main@source![])
    | move?*[s Su] = (migrate to s (print!(+$ a " has moved here.") | done@Su![]))
    | main![])
```

The program launches summoners and PA agents dynamically, using the standard migration primitive, onto the list of active sites. A function `spawnSummoner` takes the name of the site where the summoner will be spawned and the name of computer on which the graphics window will be displayed.

```
def spawnSummoner [s:Site workstation:String] =
(
  agent Summoner =
  (
        ...
    def open_window () : Status = (...)
    def main stat:Status =
        ...
    migrate to s
    run print!"Summoner installed."
    main!(open_window)
  )
  in ()
)
```

A function `spawnPA` takes a textual name of the PA agent

```
def spawn a:String =
  (agent PA =
        ...
   in
     print!(+$ a " spawned."))
```

```
( spawnSummoner![s0 "glia:0.0"]
| spawnSummoner![s1 "ouse:0.0"]
| spawnSummoner![s2 "iris:0.0"]
| spawn!"Ala"
| spawn!"Kotek")
```

For simplicity the implementation uses location-independent communication throughout, despite the fact that the name server and summoners are static.

4.2 Low-Level Translation

A usable infrastructure for the PA application can only be designed in the context of detailed assumptions, both about the system properties (e.g. the size of the network and reliability) and about the expected behaviour of the high-level agents. The PA application also demands disconnected operation (on laptops) and a higher level of fault-tolerance. We discuss infrastructure design addressing these, in [Woj00a], but for the sake of a clear example infrastructure we neglect them here. Below we describe one of the simplest algorithms possible, with a centralized server daemon. The algorithm assumes a large essentially-reliable LAN. It has been chosen to illustrate the characteristic features of Nomadic Pict, such as encoding of basic types and the use of channel names outside their declaration scope. Algorithms that are widely applicable to actual mobile agent systems would have to be yet more delicate, both for efficiency and for robustness under partial failure.

4.2.1 Algorithm

The algorithm involves a central daemon that keeps track of the current sites of all agents and forwards any location-independent messages to them. The daemon is itself implemented as an agent which never migrates; the translation of a program then consists roughly of the daemon agent in parallel with a compositional translation of the program. For simplicity we assume that programs are initiated as single agents, rather than many agents initiated separately on different sites. (Programs may, of course, begin by creating other agents that immediately migrate). In Chapter 5 we describe how to deal with the case when programs are split in many files.

The precise definition is given in 4.1 and Figures 4.2. Figure 4.1 defines a top-level. It takes name `P` of a user-defined program and name `hosts` of a parameter which is used to pass active sites to the program. The definition of `toplevel` involves creation of the daemon agent `D`, and an auxiliary compositional translation `{P}[a currentloc D SD]`, defined phrase-by-phrase, of `P` considered as part of the body of agent `a`, where the daemon agent `D` is assumed to be at site `SD`. The compositional translation is given in Figure 4.2. For each term `P` of the high-level language, the result `{P}[a currentloc D SD]` of the translation is a term of the low-level language.

4.2.2 The top level

Let us look first at the daemon. It contains two replicated inputs, on the `migrating` and `message` channels, for receiving messages from the encodings of agents. The daemon is multi-threaded — operations dealing with different agents are executed in parallel. The channel `loc` is used to enforce mutual exclusion between the bodies of the replicated inputs which deal with the same agent name (e.g. the daemon will block forwarding a message to agent `b` if `b` is in the middle of migration), and the code preserves the invariant that at any time there is at most one output on `loc`. The `loc` channel is also used to record the current site of an agent. The body of each replicated input begins with an input on `loc`, thereby acquiring both the lock and the site name.

Putting the daemon and the compositional encoding together, the top level translation, defined in Figure 4.1, creates the toplevel agent `top`, spawns the daemon agent `D`, waits for acknowledgement from the daemon and then initializes the lock channel `currentloc` for `top`, installs the replicated input on `deliver` for `top`, and starts the encoding of the body `{P}[a currentloc D SD]`. The daemon registers agent `top` to

```
new migrating : ^[Agent ^Site]
new message   : ^[#X [Agent ^Site] ^X X]
new deliver   : ^[#X ^X X Site]
new ack       : ^[]
new done      : ^^Site


{Agent} = [Agent ^Site]
{Site} = Site


{ toplevel P hosts }[Agent ^Site Agent Site] =
(
val s0 = (get_site 0)
val s1 = (get_site 1)
val s2 = (get_site 2)

agent top =
 (new currentloc : ^Site

  agent D =
   (print!"Server installed."
   | <top@s0>ack![] | currentloc!s0
   | migrating?*[b:Agent loc:^Site]=
       loc?s= <b@s>ack![]
   | message?*[#X [b:Agent loc:^Site] c:^X v:X]=
       loc?s= <b@s>deliver![c v s])

  in
    val SD = s0
    ack?_=
      ( currentloc!s0
      | deliver?*[#X c:^X v:X s:Site] = (<D@SD>currentloc!s | c!v)
      | (val !hosts = [s0 s1 s2]
         { P }[top currentloc D SD])))
())
```

Figure 4.1: The Top Level and the Daemon

be at site `s0`. We assume that the names of sites which are active in the system are stored in a configuration file and we use a library function `get_site` to read this file. The function returns the i-th site from the file. Chapter 5 presents alternative ways of configuring the Nomadic Pict system.

Turning to the compositional translation `{P}[a currenloc D SD]`, only three clauses are not trivial — for the location-independent output, agent creation, and agent migration primitives. We discuss each, together with their interactions with the daemon, in turn.

4.2.3 Location-independent output

A location-independent output in an agent `a` is implemented simply by using a location-dependent output to send a request to the daemon `D`, at its site `SD`, on its channel `message`:

```
{ c@b!v }e =
  (val  [a _ D SD] = e
  <D@SD>message![b c v])
```

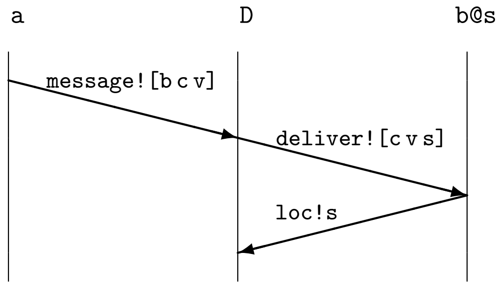
The corresponding replicated input on channel `message` in the daemon

```
{ c@b!v }e =
(val  [_ _ D SD] = e
 <D@SD>message![b c v])


{ agent b=P in Q }e =
(
  val [a loc D SD] = e
    loc?s=
      (agent B =
       (
         new currentloc : ^Site
         val !b = [B currentloc]
         ( <D@SD>currentloc!s
         | iflocal <a>done!currentloc then
             ( currentloc!s
             | {P}[B currentloc D SD])
           else ()
         | deliver?*[#X c:^X v:X s:Site] = (<D@SD>currentloc!s | c!v))
       )
       in
         done?c = (loc!s
           | (val !b = [B c]
              {Q}e)))
)


{ migrate to s P }e =
(
  val [a currentloc D SD] = e
  currentloc?_=
    ( <D@SD>migrating![a currentloc]
    | ack?_ = (migrate to s
                ( <D@SD>currentloc!s
                | currentloc!s
                | {P}e
         )))
)
```

Figure 4.2: The Compositional Translation

```
| message?*[#X [b:Agent loc:^Site] c:^X v:X]=
    loc?s= <b@s>deliver![c v s])
```

first acquires the lock and the target agent's site name `s` and sends a location-dependent message to the `deliver` channel of that agent. The lock is relinquished by the agent *after* it will receive the message. This prevents the agent migrating before the `deliver` message arrives. Note that the input on `loc` will always succeed, as the algorithm ensures that all agents register upon creation and we assume that messages are never lost. The inter-agent communications involved in delivery of a single location-independent output are illustrated below.



4.2.4 Creation

In order for the daemon's location channel `loc` to be kept up to date, agents must register with the daemon, telling it their site, both when they are created and after they migrate. Each agent records its current site internally as an output on its `currentloc` channel. This channel is also used as a lock, to enforce mutual exclusion between the encodings of all agent creation and migration commands within the body of the agent. The encoding of an agent creation in an agent `a`

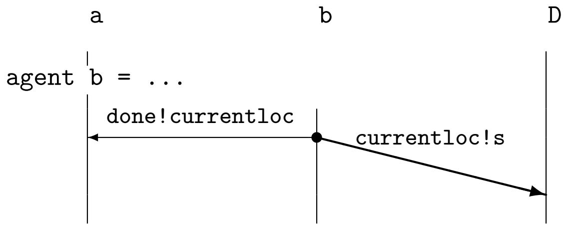
```
{ agent b=P in Q }e =
(
  val [a loc D SD] = e
  loc?s=
    (agent B =
     (
       new currentloc : ^Site
       val !b = [B currentloc]
       ( <D@SD>currentloc!s
       | iflocal <a>done!currentloc then
           ( currentloc!s
           | {P}[B currentloc D SD])
         else ()
       | deliver?*[#X c:^X v:X s:Site] = (<D@SD>currentloc!s | c!v))
     )
     in
       done?c = (loc!s
         | (val !b = [B c]
             {Q}e)))
)
```

first acquires the lock and current site `s` of `a`, and then creates the new agent `B`. The body of `B` sends a `currentloc` message to the daemon and an acknowledgement to `a` on `done` (passing the name of `B`'s `currentloc` channel). It then initializes the lock for `B` and allows the encoding of the body `P` of `B` to proceed. Meanwhile, in `a` the lock is kept until the acknowledgement from `B` is received. The name `b` from the high-level language is encoded as a pair of `B` and `currentloc`. The body of `B` is put in parallel with the replicated input

```
| deliver?*[#X c:^X v:X s:Site] = ( <D@SD>currentloc!s | c!v )
```

which will receive forwarded messages for channels in `b` from the daemon, send an acknowledgement back (on `currenloc`), and deliver the value locally to the appropriate channel.

The inter-agent communications involved in a single agent creation are illustrated below.



4.2.5 Migration

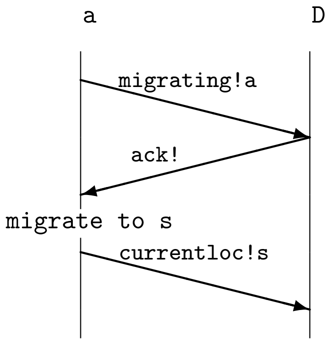
The encoding of a `migrate` in agent `a`

```
{ migrate to s P }e =
(
  val [a currentloc D SD] = e
  currentloc?_=
    ( <D@SD>migrating![a currentloc]
    | ack?_ = (migrate to s
        ( <D@SD>currentloc!s
        | currentloc!s
        | {P}e
    )))
)
```

first acquires the lock for `a` (discarding the current site data). It then sends a `migrating` message to the daemon, waits for an `ack`, migrates to its new site `s`, sends a `currentloc` message to the daemon (with the new site `s`), thereby relinquishing the lock at the daemon, and releases the local lock (also with the new site `s`). The replicated input on `migrating` in the daemon

```
| migrating?*[b:Agent loc:^Site]=
    loc?s= <b@s>ack![]
```

first acquires the lock and the current site of `a` and sends an `ack` to `a` at that site. The inter-agent communications involved in a single migration are shown below.



The whole program structure is in Figure 4.3. In our translation we have made an assumption that the application program and translation encoding are compiled and executed together, and so the program begins from a single toplevel which creates a daemon and other agents that immediately migrate if necessary. Since Nomadic Pict is thought of to be a language for prototyping this is fair enough. However, distributed programs can often be split in many files which we should be able to compile and execute on separate machines. In this case, we should include the daemon definition only for one executable and export its name (and all other names which are used to communicate with the daemon) using the library functions for trading names in the system. Other executables would have to subscribe for these names before they can use them.

```
import "Nstd/Map"
import "Graphics/Graphics"

program hosts : [Site Site Site] =
(
  ...
)

new migrating : ^[Agent ^Site]
new message   : ^[#X [Agent ^Site] ^X X]
new deliver   : ^[#X ^X X Site]
new ack       : ^[]
new done      : ^^Site

{Agent} = [Agent ^Site]
{Site} = Site

{ toplevel P hosts }[Agent ^Site Agent Site] =
(
  ...
)
{ c@b!v }e = ...
{ agent b=P in Q }e = ...
{ migrate to s P }e = ...
```

Figure 4.3: The Program Structure

Chapter 5

Compilation and Execution

5.1 To Get Started

To execute a Nomadic Pict program, e.g.

```
run print!"Hello world!"
```

first place it in a file named with a `.pi` suffix, e.g. `prog.pi`, and then compile and execute this file by running the Nomadic Pict compiler as follows:

```
np prog.pi
```

This spawns a Nomadic Pict virtual machine on your current machine which will execute the program. If your program contains only the Pict language, you may prefer to choose a standard Pict compiler as follows

```
np -set cc prog.pi -o prog
```

or simply `pict prog.pi -o prog`, and execute `prog`. The native code generated by the Pict compiler is much faster then the code interpreted by the Nomadic Pict virtual machine but then you cannot use the primitives which are characteristic for Nomadic Pict.

5.2 Separate Compilation

The compiler provides a simple facility for breaking up large programs into parts (modules), storing the parts in separate files, and compiling these files separately. In the beginning of a program you can write

```
import "name"
```

where `"name"` is an absolute or relative pathname (not including the suffix `.pi`) of a separately compiled module to be imported. The module will be included at the point where the first `import` for this module name appears. If a relative pathname is used, both the current directory and a central directory of Nomadic Pict library files are searched.

Before a file can be imported by other files, it must be compiled by the compiler to yield a file with suffix `.px`, e.g. this below

```
np -set sep prog.pi -o prog.px
```

produces an object file `prog.px` and does *not* spawn a runtime system to execute it. Then we can import the module in other programs using `import "prog"` and compile as before. A few basic library modules are imported by default.

The Pict libraries of precompiled modules often contain C procedures and so they cannot be directly used in programs which are interpreted by the Nomadic Pict virtual machine (one can, of course, use them in programs which are compiled to the native code). Therefore we tried to support in the Nomadic Pict virtual machine all Pict core libraries and all those which are often used (more libraries may be ported in the future). The new libraries have the same names and interfaces — just a different path need to be specified in the `import` declaration. A rule of thumb is that if an original Pict library has name `Path/Name` then the library which is recognised by the virtual machine will have a name `Npath/Name` (this does not work for libraries which are supported by the Nomadic Pict system only — they may have arbitrary names).

The current distribution includes libraries implementing a variety of data structures, interfaces to operating system and the Nomadic Pict runtime system, and some experimental services, such as distributed events. Those libraries which use calls to the Nomadic Pict runtime system or contain any constructs which are characteristic for Nomadic Pict will not compile using the original Pict compiler. All libraries supported by the Nomadic Pict virtual machine are described in full in [Woj00b].

Unfortunately, the standard libraries from the Nomadic Pict distribution can only use Low-Level Nomadic Pict (and so only distributed idioms which are location-aware). This is a result of early design decisions to put the translation from High- to Low-Level Nomadic Pict at the top of the compiler architecture. The modules of libraries are precompiled to an intermediate code which can be linked with user-defined programs. The intermediate code is fairly low-level and is linked *after* the definitions of the translation from High- to Low-Level Nomadic Pict are applied in programs. The rest of this chapter mostly deals with distributed programming in the Nomadic Pict system.

5.3 Language Translations

If our program does not use any high-level language primitives then it can be organised simply as a file containing a sequence of declarations preceded by a number of `import` clauses. Otherwise, we need to structure the program as follows

```
import "name"
{- other imports and any global declarations -}

program par : T =
(
  {- a user-defined program in the high-level language -}
)

{-*****************************************************************

   any global declarations of the compositional translation here -}

{toplevel par} T' =
(
  {- top-level definition here -}
)
{- a compositional translation of types and primitives -}
```

After imports we can have any global declarations such as constants and global functions which we want to use in a program *and* the compositional translation. Then we can define an actual program in the high-level language using the `program` construct.

Following the star line are declarations of the compositional translation (they can be stored in a separate file if required). Firstly, we declare any global constants, functions, and channel names which are used by the definitions of the compositional translation, then we define a top-level (in the low-level language) using `toplevel`. The top-level defines the top-level actions such as spawning the distributed infrastructure which have to be executed before the high-level program starts. Finally, we have definitions of the compositional translation of all interesting types and high-level prmitives. If some definition is missing, the compiler will replace a high-level primitive by its direct equivalent in the low-level language if it exists; the location-independent output `c@a!v` will be replaced by `iflocal <a>c!v then () else ()`.

The Nomadic Pict distribution contains a few example distributed infrastructure encodings (each in one file). They can be included in user-defined programs below the star line as above, with just minor changes in the top-level declaration. The top-level of each infrastructure encoding defines a tuple of a few (usually 2-3) active sites which is passed to the user-defined program via the parameter `par` of type `T = [Site ... Site]`. This parameter is likely to be customized to accommodate the real number of sites which are going to be used. This may sometimes require to analyse the code of the top-level and, e.g. spawn additional daemons on sites, etc. In the end of this chapter, we describe how to pass real machine and port numbers to Nomadic Pict programs, and configure the distributed Nomadic Pict runtime system to run on many machines.

More changes in the translations are required if a user-defined program is scattered in many files to be compiled on different machines. In this case, we have to copy the top-level and the compositional translation in all files, modifying the top-level accordingly, e.g. selecting one to be a server and other the clients, etc. We also have to trade names and values in the distributed system so that different parts of the compositional translation can communicate using the same names of channels and agents. Below we describe the idea, see also example programs included in the Nomadic Pict distribution for details.

5.4 Trading Names and Values

Nomadic Pict has been designed as a language for prototyping distributed applications and we almost never needed to split programs in many files which are compiled and executed separately on different machines. We were simply spawning different parts of distributed programs dynamically on "empty" Nomadic Pict runtime systems, using agents and migration. However, occasionally it is convenient to compile and execute server and client programs (likely to be on different machines) separetly and at different time, e.g. in demo programs.

The `Nstd/Sys` library offers two functions `publish` and `subscribe` that can be used in order to exchange names, basic values, and any complex values which can be sent along channels at runtime, thus making possible to set up connection between different programs. Below is an example program which is split into files `server.pi` and `client.pi`.

```
{- server.pi -}
new c : ^String
val s = (this_site)
agent b = ((publish "foo" (dynamic [b s c]));
           c?p= print!p)
```

In file `server.pi`, the program creates a new channel name `c`, assigns the current site name to `s`, creates agent `b` and publishes a record containing `c`, `s`, and `b` at the system trader. After the names are published, the program waits for a message on `c` and prints the message out. The function `publish` takes as arguments a value to be published (which must be converted to a type `Dyn`) and a string keyword to identify the value.

```
{- client.pi -}
agent a =
  typecase (subscribe "foo" a) of
    [ag:Agent si:Site ch:^String] ->
      <ag@si>ch!"Hello world!"
  else print!"Type mismatch for foo"
```

In file `client.pi`, the program creates agent `a` and subscribes for the value published in file `server.pi`. The function `subscribe` takes two parameters: the string keyword `"foo"` which was used to publish the value at the trader, and the name of the current agent. The function blocks until the value is available. The value returned by `subscribe` is a dynamic value which can be matched against expected types using `typecase`. If the dynamic typechecking succeeds then basic values extracted from the dynamic value can be used for communication (e.g. in the program above we send a message on `c` to agent `b` which is supposed to be at `s`).

When the runtime system starts up, we have to specify — using options `-trader` and `-tport`, an address and port number for the runtime system selected to be a trader. By default the current runtime system is chosen. For example, if we compile `server.pi` on `glia.cl.cam.ac.uk` as below

```
np server.pi
```

then the local machine `glia` will be selected to trade names subscribed or published by any local `subscribe` and `publish` function calls. The execution of `nc` will display a local port number selected by the runtime system, e.g. `5000`. We can then use this information and execute the client program on another machine as follows

```
np client.pi -trader glia.cl.cam.ac.uk -tport 5000
```

This will select the runtime system on `glia.cl.cam.ac.uk`, listening on a port `5000`, to be a trader, and execute the `client` program on the local machine.

The functions `publish` and `subscribe` will almost certainly change in the future releases of Nomadic Pict to accommodate possible inavailablity of the trader (e.g. during the disconnected operation on laptop computers); currently we assume that there is connection with the trader when performing these functions.

5.5 Configuring the System

The names of sites can be either obtained from a local configuration file or a trader, using different library functions. Below we describe both methods.

5.5.1 Using a config file

The configuration file should contain pairs of DNS names (enclosed in double-quotes) and port numbers of the active sites that we want to use in our programs; we can use `#` to comment lines. For example, a valid configuration file may look as follows

```
"glia"     5002
"vesicle"  5001
#"britten" 5000
"iris.cl.cam.ac.uk" 5003
#"puccini" 5000
```

The configuration file has a default name `config`; the compiler's option `-f` can be used to name other file. When the runtime system starts up on a local machine, it opens the `config` file (or the file specified by option `-f`) and looks up for the first occurance of the Internet address of the local machine, choosing the port name given next to the address.

We can use a library function `get_site i`, where $i = 0, 1, ...$, to return a site name built using the IP address and port number from the i-th valid line of the `config` file (commented lines are ignored). Another useful library function is `this_site`, which returns the Nomadic Pict name of the current site (or more precisely of the current instantiation of the runtime system on the local machine).

In order to execute a distributed program on many machines, we must run a runtime system on each machine. The execution of a program spawns a runtime system on a local machine. We can start an "empty" runtime system on other machines using `nc` without specifying a program name as follows

```
np -f config_file
```

or simply

```
np
```

Using the `config` file is good enough for quick prototyping. The translation encodings in the Nomadic Pict distribution usually assume this method and use the function `get_site` to obtain site names from a local file. This, however, can be easily modified. Below we describe using a trader mechanism which is more elegant.

5.5.2 Using a trader

If the runtime system cannot find the configuration file in a local directory then it will select a free port number automatically, create a new fresh name of the site, and publish it with the "dotted" symbolic name of the local host machine, e.g. `sinapsi.cl.cam.ac.uk`, as a keyword (to be precise this is a string name returned by a Unix function `gethostname`). If we type

```
np
```

to start an 'empty'" runtime system then a local trader will be selected by default (and a port number of the runtime printed out). We can choose another trader using the options `-trader` and `tport`, e.g.

```
np -trader <host_address> -tport <port_number>
```

Then, any programs executed on other machines can use a function `subscribe_site` to obtain the site name from the trader, specifying a full "dotted" symbolic name of the host which runs the site. This below

```
agent top =
(
  val s = (subscribe_site <host_address> top
  ...
)
```

returns the name `s` of type `Site` of the runtime system running on `sinapsi.cl.cam.ac.uk` and can be used later in the program (each execution of `subscribe_site` may involve network communication, so it should be executed only once for each site, e.g. in the beginning of the program). Note that this mechanism assumes only one runtime system per machine — this however can be changed by modifying the bootstrapping file, described below.

The user-defined programs must be compiled with options `-trader` and `-tport`, so that the runtime system will know which trader should be contacted, e.g.

```
np prog.pi -trader <host_address> -tport <port_number>
```

5.5.3 Bootstrapping the system

Actually, if we do not specify any program name, the directive `nc` will execute a default bootstrapping program defined in a system file `bootstrap.pi` (to find the file first the current directory will be search then a central directory of Nomadic Pict libraries)

```
(publish_this_site (sys.gethostname)); (pr "Ready...");

new foo : ^[]
agent a =
  foo?*_ = ()
```

The program publishes the site name of the local host identified by the Internet address of the local host machine and executes an empty loop. Additional behaviour can be added if required by modifying this file.

Chapter 6

Syntax

This chapter describes the syntax of Nomadic Pict programs (for description of lexical rules and Pict syntax we use extracts from [PT97b], by courtesy of Benjamin C. Pierce).

6.1 Lexical Rules

Whitespace characters are space, newline, tab, and formfeed (control-L). Comments are bracketed by `{-` and `-}` and may be nested. A comment is equivalent to whitespace.

Integers are sequences of digits (negative integers start with a `-` character). Strings can be any sequence of characters and escape sequences enclosed in double-quotes. Sites can be any sequence of characters and escape sequences enclosed in double single-quote characters (`''`), used to denote the IP address, followed by a colon and integer, to denote a port number. The escape sequences `\"`, `\n`, and `\\` stand for the characters double-quote, newline, and backslash. The escape sequence `\ddd` (where `d` denotes a decimal digit) denotes the character with code `ddd` (codes outside the range `0..255` are illegal). Character constants consist of a single quote character (`'`), a character or escape sequence, and another single quote.

Alphanumeric identifiers begin with a symbol from the following set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Subsequent symbols may contain the following characters in addition to those mentioned above:

```
0 1 2 3 4 5 6 7 8 9 '
```

Symbolic identifiers are non-empty sequences of symbols drawn from the following set:

```
~ * % \ + - < > = & | @ $ , `
```

6.2 Reserved Words

The following symbols are reserved words:

```
Agent      agent    and     Bool    ccode     Char       def      dynamic  else   false
if         iflocal  import  inline  Int       in         migrate  new      now    of
program    rec      run     Site    String    terminate  then     timeout  to     Top
toplevel   true     Type    type    typecase  val        switch   wait     where  with
@          ^        \       /       .         ;          :        =        |      !
#          ?        ?*      _       <         >          ->       {        (      [
}          )        ]
```

6.3 Concrete Syntax

For each syntactic form, we note whether it is part of the core language (C), the language for expressing encodings (T), a derived form (D), an optional type annotation that is filled in during type reconstruction if omitted by the programmer (R), or an extra-linguistic feature (E). Syntactic forms characteristic for the Nomadic Pict language are marked by n.

6.3.1 Compilation units

TopLevel	=	*Import* ... *Import* *Dec* ... *Dec*	*E*	Compilation unit
		Import ... *Import* *TopDec* ... *TopDec*	*En*	Compilation unit
Import	=	`import` *String*	*E*	Import statement

6.3.2 Top-level declarations

TopDec	=	*Dec*		Declaration
		`{` *Agent* `} =` *Type*	*Tn*	Agent type
		`{` *Site* `} =` *Type*	*Tn*	Site type
		`program` *Id* `:` *Type* `=` *Proc*	*Tn*	Program declaration
		`{ toplevel` *Id* *Id* `}` *Type* `=` *Proc*	*Tn*	Toplevel declaration
		`{ def` *Id* `}` *Id* *Abs*	*Tn*	Process abstraction
		`{ agent` *Id* `=` *Id* `in` *Id* `}` *Id* `=` *Proc*	*Tn*	Agent creation
		`{ migrate to` *Id* *Id* `}` *Id* `=` *Proc*	*Tn*	Agent migration
		`{` *Id* `?*` *Id* `=` *Id* `}` *Id* `=` *Proc*	*Tn*	Replicated input
		`{ <` *Id* `@` *Id* `>` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Output to agent on site
		`{ <` *Id* `>` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Output to adjacent agent
		`{ iflocal <` *Id* `>` *Id* `!` *Id* `then` *Proc* `else` *Proc* `}` *Id* `=` *Proc*	*Tn*	Test-and-send to agent
		`{` *Id* `@` *Id* `!` *Id* `}` *Id* `=` *Proc*	*Tn*	Location-independent output
		`{` *do* *String* *Id* `in` *Id* `}` *Id* `=` *Proc*	*Tn*	Macro definition

6.3.3 Declarations

Dec	=	`new` *Id* `:` *Type*	*C*	Channel creation
		`val` *Pat* `=` *Val*	*D*	Value binding
		`run` *Proc*	*D*	Parallel process
		Val `;`	*D*	Sequential execution
		`inline def` *Id* *Abs*	*D*	Inlinable definition
		`def` Id_1 Abs_1 `and ... and` Id_n Abs_n	*C*	Recursive definition ($n \geq 1$)
		`type` *Id* `=` *Type*	*D*	Type abbreviation
		`type (` *Id* $KindedId_1 \ldots KindedId_n$ `) =` *Type*	*D*	Type operator abbrev ($n \geq 1$)
		`now (` *Id* *Flag* ... *Flag* `)`	*E*	Compiler directive
		`agent` Id_1 `=` $Proc_1$ `and ... and` Id_n `=` $Proc_n$	*Cn*	Agent creation ($n \geq 1$)
		`agent` Id_1 `=` $Proc_1$ `and ... and` Id_n `=` $Proc_n$ `in`	*Cn*	Agent creation ($n \geq 1$)
		`migrate to` *Val*	*Cn*	Migrate to site
		`do` *String* *Val*	*Tn*	Macro inlining
		`do` *String* *Val* `in`	*Tn*	Macro inlining
		`{` *Id* `}` *Val*	*Tn*	Declaration inlining

Flag	=	*Id*	*E*	Ordinary flag
		Int	*E*	Numeric flag
		String	*E*	String flag

6.3.4 Abstractions

Abs	=	*Pat* = *Proc*	C	Process abstraction
		(*Label FieldPat* ... *Label FieldPat*) *RType* = *Val*	D	Value abstraction

6.3.5 Patterns

Pat	=	*Id RType*	C	Variable pattern
		[*Label FieldPat* ... *Label FieldPat*]	C	Record pattern
		(rec *RType Pat*)	C	Rectype pattern
		_ *RType*	C	Wildcard pattern
		Id RType @ *Pat*	C	Layered pattern
		! *Id*	T	Reference pattern
FieldPat	=	*Pat*	C	Value field
		# *Id Constr*	C	Type field

6.3.6 Type constraints

Constr	=	⟨*empty*⟩	D	No constraint
		< *Type*	C	Subtype constraint
		= *Type*	C	Equality constraint

6.3.7 Processes

Proc	=	*Val* ! *Val*	C	Output atom
		Val ? *Abs*	C	Input prefix
		Val ?* *Abs*	Cn	Replicated input
		wait *Val* ? *Abs* timeout *Val* -> *Proc*	Cn	Timed input
		< *Val* @ *Val* > *Val* ! *Val*	Dn	Output to agent on site
		< *Val* > *Val* ! *Val*	Dn	Output to adjacent agent
		iflocal < *Val* > *Val* ! *Val* then *Proc* else *Proc*	Cn	Test-and-send to agent
		Val @ *Val* ! *Val*	Dn	Location-independend output
		( )	C	Null process
		$(Proc_1 \mid \ldots \mid Proc_n)$	C	Parallel composition ($n \geq 2$)
		($Dec_1 \ldots Dec_n$ *Proc*)	C	Local declarations ($n \geq 1$)
		if *Val* then *Proc* else *Proc*	C	Conditional
		terminate	C	Agent termination
		typecase *Val* of Pat_1 -> $Proc_1$ ... Pat_n -> $Proc_n$ else $Proc_{n+1}$	Dn	Type matching ($n \geq 1$)
		switch *RType Val* of (Id_1 > Pat_1 -> $Proc_1$ ... Id_n > Pat_n -> $Proc_n$)	Dn	Variant matching ($n \geq 1$)
		{ *Id* } *Val*	Tn	Process inlining

6.3.8 Values

Val	=	*Const*	*C*	Constant
		Path	*C*	Path
		`\` *Abs*	*D*	Process abstraction
		`[` *Label FieldVal* ... *Label FieldVal* `]`	*C*	Record
		`if` *RType Val* `then` *Val* `else` *Val*	*D*	Conditional
		`(` *Val RType* `with` *Label FieldVal* ... *Label FieldVal* `)`	*D*	Field extension
		`(` *Val RType* `where` *Label FieldVal* ... *Label FieldVal* `)`	*D*	Field override
		`(` *RType Val Label FieldVal* ... *Label FieldVal* `)`	*D*	Application
		`(` *Val* `>` $Val_1 \ldots Val_n$ `)`	*D*	Right-assoc application ($n \geq 2$)
		`(` *Val* `<` $Val_1 \ldots Val_n$ `)`	*D*	Left-assoc application ($n \geq 2$)
		`(` `rec` *RType Val* `)`	*C*	Rectype value
		`(` $Dec_1 \ldots Dec_n$ *Val* `)`	*D*	Local declarations ($n \geq 1$)
		`(` `ccode` *Int Id String FieldVal* ... *FieldVal* `)`	*E*	Inline C code (Pict only)
		`(` `ccode` *Int Id String FieldVal* ... *FieldVal* `)`	*En*	System function call
		`(` `dynamic` *Val RType* `)`	*Dn*	Typed value
		`[` *Id* `>` *Val* `]`	*Dn*	Variant
		`typecase` *RType Val* `of` Pat_1 `->` Val_1 ... Pat_n `->` Val_n `else` Val_{n+1}	*Dn*	Type matching ($n \geq 1$)
		`switch` *RType Val* `of` `(` Id_1 `>` Pat_1 `->` Val_1 ... Id_n `>` Pat_n `->` Val_n `)`	*Dn*	Variant matching ($n \geq 1$)
		`{{` *Id* `}}`	*Tn*	Value inlining
Path	=	*Id*	*C*	Variable
		Path `.` *Id*	*C*	Record field projection
FieldVal	=	*Val*	*C*	Value field
		`#` *Type*	*C*	Type field
Const	=	*String*	*C*	String constant
		Char	*C*	Character constant
		Int	*C*	Integer constant
		`true`	*C*	Boolean constant
		`false`	*C*	Boolean constant

6.3.9 Types

Type	=	`Top`	C	Top type
		Id	C	Type identifier
		`^` *Type*	C	Input/output channel
		`!` *Type*	C	Output channel
		`/` *Type*	C	Responsive output channel
		`?` *Type*	C	Input channel
		`Int`	C	Integer type
		`Char`	C	Character type
		`Bool`	C	Boolean type
		`String`	C	String type
		`[` *Label FieldType* ... *Label FieldType* `]`	C	Record type
		`(` *Type* `with` *Label FieldType* ... *Label FieldType* `)`	D	Record extension
		`(` *Type* `where` *Label FieldType* ... *Label FieldType* `)`	D	Record field override
		`\` $KindedId_1 \ldots KindedId_n$ `=` *Type*	C	Type operator ($n \geq 1$)
		`(` *Type* $Type_1 \ldots Type_n$ `)`	C	Type application ($n \geq 1$)
		`(` `rec` *KindedId* `=` *Type* `)`	C	Recursive type
		`Agent`	Cn	Agent type
		`Site`	Dn	Site type
		`Dyn`	Dn	Dynamic type
		`[` Id_1 `>` $Type_1$ ... Id_n `>` $Type_n$ `]`	Dn	Variant type
		`{` *Id* `}`	Tn	Type inlining
FieldType	=	*Type*	C	Value field
		`#` *Id Constr*	C	Type field
RType	=	⟨*empty*⟩	R	Omitted type annotation
		`:` *Type*	C	Explicit type annotation

6.3.10 Kinds

Kind	=	`(` $Kind_1 \ldots Kind_n$ `->` *Kind* `)`	C	Operator kind ($n \geq 1$)
		`Type`	C	Type kind
KindedId	=	*Id* `:` *Kind*	C	Explicitly-kinded identifier
		Id	D	Implicitly-kinded identifier

6.3.11 Labels

Label	=	⟨*empty*⟩	C	Anonymous label
		Id `=`	C	Explicit label

Bibliography

[BBHM95] J. M. Bacon, J. Bates, R. J. Hayton, and K. Moody. Using events to build distributed applications. In *Proceedings of SDNE '95*, 1995.

[Bou92] Gérard Boudol. Asynchrony and the pi-calculus. Technical Report RR-1702, Inria, Institut National de Recherche en Informatique et en Automatique, 1992.

[Hay96] Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services.* PhD thesis, University of Cambridge, 1996.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[Nee89] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. Addison-Wesley, 1989.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Theory and Practice of Parallel Programming (TPPP, Sendai, Japan, 1994)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer Verlag, 1995.

[PT97a] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. Appeared in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 2000.

[PT97b] Benjamin C. Pierce and David N. Turner. *Pict Language Definition*, 1997. Available electronically as part of the Pict distribution.

[PT97c] Benjamin C. Pierce and David N. Turner. *Programming in the Pi-Calculus. A Tutorial Introduction to Pict*, 1997. Available electronically as part of the Pict distribution.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages (ICCL '98 Workshop, Chicago, USA, May 1998)*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1999. Also appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[Woj00a] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, University of Cambridge, 2000. Also appeared as Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.

[Woj00b] Paweł T. Wojciechowski. *Nomadic Pict Language Libraries*, 2000. Available electronically as part of the Nomadic Pict distribution.