PTOM – A PASCAL Translator For Mobile Code

Christian F. Tschudin

Technical Report 96.06, July 1996

Computer Science Department, University of Zurich,
Winterthurerstr. 190, CH-8057 Zurich, Switzerland

Email: tschudin@ifi.unizh.ch

Abstract

We report on an implementation of a PASCAL translator that generates code for mobile software messengers (agents): these agents can roam a network of interconnected execution platforms on behalf of a user, enabling users to implement customized network functionality themselves. In this report we focus on the language extensions (PASCAL-M) and the implementation aspects of the translator. The main departure in PASCAL-M from standard PASCAL is that the programmer has access to the generated code in form of string constants. An interesting aspect of the "PASCAL-to-M∅" (PTOM) translator is that PASCAL-M is translated to the stack-based high-level language M∅. M∅ is similar to POSTSCRIPT and has no notion of memory addresses although PASCAL compilers usually produce code for machines or software interpreters with word addressable memory: because M∅ completely hides the exact memory locations of code as well as data, we have to use other implementation approaches for activation frames, variable dereferencing and code fixups. Other aspects of the translator implementation like data mapping and heap management are also described. Finally, we relate our work to recent developments in the field of intermediate code and languages.

Keywords: PASCAL, M∅, POSTSCRIPT, communications messengers, mobile software agents.

1 Introduction

PASCAL is still one of the most used languages when it comes to explain algorithms in textbooks or scientific journals. In our research project on mobile code we also had the need to document the algorithms we used for programming the network with "communications messengers" i.e., small autonomous mobile code pieces that can roam the network. Therefore, we developed a new PASCAL dialect – PASCAL-M – and the PTOM translator that translates PASCAL-M to the target language M∅. The novel element of PASCAL-M is that the programmer has direct access to the generated code from within PASCAL-M. This allows for explicit manipulation of code strings to be shipped to neighboring messenger execution platforms.

1.1 Mobile Code and the Language Issue

Mobile software agents are based on the concept of remote programming. In principle, this concept could be implemented by the exchange of code only (mobile code), together with a mechanism at the receiver side that accepts incoming code and executes it[1]. For portability reasons one uses intermediate code instead of native code. Programs expressed in such a code are either interpreted or compiled on-the-fly to the receiver's native instruction set. A third language level may provide more comfort for the programmer: agents can then be expressed in an object oriented high-level language which is compiled to intermediate code and then interpreted or further compiled into native code. In section 5.1 we point to other programming languages used for mobile code.

Which mobile code language is chosen depends on many (external) factors, including personal preferences. The `MØ` language was developed in a bottom-up approach as a fast lane to a running mobile code system. On one hand it adheres to an approach of pure mobile code in trying to minimize the assumptions about and the operational effort for a mobile code infrastructure: mobile code should be used to *built* such infrastructures instead of relying on their existence. On the other hand, `MØ` was chosen to be a high-level language making the programming task fast and less error prone. `MØ` is a stack-based postfixed language with a very compact syntax that is presented in section 2. The mobile code fragments expressed in `MØ` are called *messengers* (which nicely contrasts with the more passive concept of "messages").

Perhaps even more important that the comfort of programming is the expressive power and the underlying execution model of a programming language. A conceptual argument in `MØ` is that remote programming inevitably involves late binding: when a messenger (mobile agent) arrives at a remote system, its sequence of instructions has to be bound to the local instruction set and/or to subroutines already resident at the remote system. Another issue of `MØ` is that we explicitly envisage metaprogramming i.e., the binding to procedures not known at compiletime and the generation of new messengers at runtime from within `MØ` code. Both concerns, binding at receive time and metaprogramming, led to disregard languages that require compilation, although (transparently) compiling `MØ` could be a possible optimization in the future.

When we consider PASCAL as a mean of expressing messengers we have conflicting interests because PASCAL's type checking is contrary to `MØ`'s late binding principle. PASCAL checks all types at compile time as many other modern languages do: mobile agent systems built on top of such languages naturally try to extend type checking to the programming of agents. An outcome of extending PASCAL in order to support mobile agents is that it makes the different underlying concepts and implicit assumptions visible and also permits a direct comparison with other mobile code systems based e.g., on JAVA. In our case we can relate PASCAL-M and `MØ` to JAVA and the bytecodes for the JAVA virtual machine, respectively. Note however that PTOM is a translator, not a compiler, because the target language is not native CPU code. In fact, PTOM "just" expresses programs written in a high-level language in some other high-level language.

The latest PASCAL source code of the PTOM translator (including a few PASCAL-M programming examples) and this documentation are accessible under

[1]However, in practice one often makes a distinction between an agent's code and the agent's state that has to be transferred. Also, there are often some restrictions on which code should be accepted and executed by means of mutual authentication of agent and execution platform.

`ftp://ftp.ifi.unizh.ch/pub/projects/msgr/ptom/`
This technical report is also online available as
`ftp://ftp.ifi.unizh.ch/pub/techreports/TR-96/ifi-96.06.ps.gz`

1.2 Overview

In section 2 of this report we give a brief introduction to the messenger concept and the target language `MØ` that is understood by all messenger execution platforms. For a full account of the `MØ` language we refer to [18], as well as we assume in this paper some familiarity with the execution model and data types of POSTSCRIPT [1]. Section 3 introduces the extensions of PASCAL-M while in section 4 we cover some implementation details of the translator proper and show an example program written in PASCAL-M. Section 5 concludes this report with a discussion and references to related work, mainly the recent growing interest in intermediate languages and intermediate code representation for compilers due to the advent of mobile code.

2 Messengers and the Programming Language `MØ`

"Communications by messengers" is an alternate paradigm for implementing computer communications [17]: instead of exchanging predefined messages according to a well-defined protocol between pre-installed protocol entities, protocols are implemented with mobile code fragments only. These messengers carry with them the logic of the protocol to implement. The advantage of this approach is that protocol-specific software must not be installed beforehand of any communication between distant hosts – binding a communications service to an implementation is delayed until runtime and ruled by the initiator. The price for the protocol genericity and the increased flexibility is that the range of optimization is narrowed (once a ordinary message-exchange protocol has proven to be useful, highly optimized implementations are deployed and statically installed). The messenger paradigm can also be characterized as the *instructional* form of computer communications while classical message exchange is based on message *interpretation*.

For the operation of a messenger-based communication architecture one needs *execution platforms* which accept arbitrary messengers and execute them. At the transmission level, we have *messenger packets* sent to platforms that consist mainly of a sequence of instructions. Each received messenger packet becomes an active *messenger process* that is unconditionally executed. It can locally interact with the execution platform and is able to submit new messenger packets to other platforms. In the sequel we use the term *messenger* for messenger packets as well as messenger processes.

`MØ` is a first implementation of the messenger paradigm [19]. It defines a data format for messenger packets and specifies the internal structure (execution model) of an `MØ` execution platform. `MØ` is also the name for the language in which the messengers' instructions are expressed. Figure 1 shows the main elements of the `MØ` messenger execution platform.

- Messenger packets received by the platform become concurrent and independent *processes*.

- Messenger processes interact through a *shared memory* area: messengers can deposit data values under arbitrary keys. In `MØ`, this is realized as a public

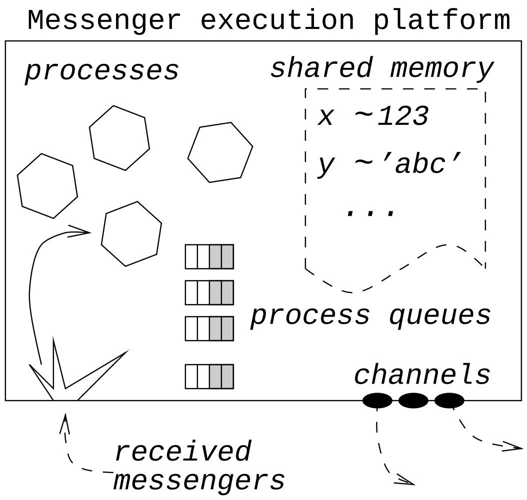

Figure 1: The structure of a messenger execution platform.

"dictionary" that all processes can read from and write to (but which is not browsable).

- Messenger processes can synchronize their execution with *process queues*: if a process enters a queue (which is identified by a 64-bit key), it becomes active only when it advances at the queue's head position. A process can be in only one queue at any time: entering another queue automatically removes the process from its current queue. Queues can be stopped (inhibiting other queued processes to advance) and later be restarted.

- Messenger processes can create new messenger packets and request their transfer to neighboring platforms with the *submit* command. The messenger transfer is not reliable i.e., messengers may be lost or be delivered in desequenced order. It is also possible to spawn local messenger processes.

- Finally note that a platform offers, except the transmission to neighboring platforms, only *local* services. All communication services that would involve the cooperation of several platforms must be implemented by messengers.

Messengers are prototypic mobile agents in the sense that they represent the essence of mobile code. Using the simple messengers one can built richer and perhaps even intelligent mobile agents, including direct agent-to-agent communications.

2.1 Comparing `M∅` with POSTSCRIPT

The `M∅` programming language is structurally very similar to POSTSCRIPT although graphics related operators are completely missing. There is an operand stack, a stack of dictionaries for name resolution and a (hidden) execution stack maintained by the `M∅` interpreter. There are simple data values (integers, time values, 64-bits keys and names) and container types (arrays, strings, dictionaries and FIFO message queues). Many standard operators of POSTSCRIPT are also present in `M∅`. This is also true for the construction of arrays and procedures with the mark and array operators `[ ]` `{}`, respectively. `M∅`'s submission channels and process queues are opaque types that can not be directly accessed: instead, they must be referenced by keys which are a kind of handle. At the lexical level, the differences between M∅ and POSTSCRIPT are more prominent:

- Uppercase names are limited to a single character. Most standard operators of M∅ have uppercase names, which leads to a very compact coding because no delimiting space characters are required. Example:
  (POSTSCRIPT) `currentdict{exch exch pop pop}forall`
  (M∅) `.{XXPP}L`
  Multicharacter identifiers thus are limited to lowercase characters and may not contain punctuation characters which are also defined to be single character identifiers.

- Strings are in M∅ delimited with double quotes instead of parenthesis. The backslash can be used to define strings by giving hex digits: `"xyz"` is the same as `\78797a\`.

- To quote names POSTSCRIPT uses the slash character, which in M∅ becomes the quote character `'`.

Many M∅ and POSTSCRIPT operators – beside differing in names – behave indenticaly. Figure 2 gives the equivalences for some of the standard M∅ operators.

3 The PASCAL-M Language

PASCAL-M is a new PASCAL dialect for expressing messenger code. The reason for defining yet another PASCAL was twofold: on one hand, we realized that messenger expressed in M∅ are hardly "self–documenting" which makes it difficult to exchange ideas about mobile code by showing code examples. It is also true that programming in M∅ is tedious and M∅ code should better be generated by programs. On the other hand, we think that programming with mobile code is not mastered yet as well as discussions about the right way to support it at the language level are going on in the object–oriented community. Our view is that mobile code questions some very basic assumptions on "objects" like e.g., the stationary flow of control and the consequently stationary method interface (in Agha's actor model, this translates to global mailbox addresses). In order to avoid an investigation into problems of object–orientation instead of mobile code, we adopt a minimalistic approach: messengers are lean mobile flows of control with private memory that can roam the network and which can enter into contact only through side effects in a platform's shared memory. Wirth's classic PASCAL [11] is therefore a good starting point for developing a high-level language for these purely sequential messengers. Our viewpoint is that PASCAL-M is an experimental language that should help to better understand programming requirements for mobile code.

3.1 Composing Messengers at Run Time

One central element of the messenger programming language M∅ is the ability of messenger processes to compose other messengers. While in high-level languages like PASCAL or C one has a clear separation between high-level language and machine code, this separation becomes impeding with messengers. In fact arbitrary character strings can in M∅ be claimed to be the code of a new messenger (in case of invalid instructions, missing or wrong parameters, the messenger is simply aborted). At

M0	POSTSCRIPT	comment
`.`	`currentdict`	push topmost dictionary
`[`	`[`	push a mark
`]`	`]`	collect all elements downto mark into an array
`{`	`{`	push a procedure mark
`}`	`}`	collect all elements downto mark into a procedure array
`:`	`put`	define an dictionary entry or set an array element
`;`	*no equivalent*	push a new random key
`_`	*no equivalent*	(underscore) name of the global dictionary that represents the shared memory area
`?`	`ifelse`	execute one of the two elements according to the boolean value
`'`	`/`	quote character
`(`	`begin`	move dictionary from operand to dictionary stack
`)`	`end`	discard topmost element of dictionary stack
`A`	`array`	create an array of given size
`D`	`dict`	create a dictionary (in `M0` no size must be given)
`F`	`find`	find the topmost dictionary where the given name is defined
`G`	`get`	get a dictionary entry or an array element
`H`	`stopped`	execute a procedure in a new context and catch errors
`I`	`index`	push a copy of the ith element relative to the operand stack's top
`J`	`exit`	jumps out of the innermost loop
`L`	`for` and `loop`	loops a given number of times (or indefinitely)
`L`	`forall`	loop over an array or dictionary
`P`	`pop`	discard the topmost element of the operand stack
`S`	`string`	create a string instance of given length
`X`	`exch`	swap the two topmost operands
`Z`	`stop`	terminate the inner-most `H` context

Figure 2: Important `M0` operators and their POSTSCRIPT equivalent.

a first look, PASCAL-M respects the classical "high-level/machine-level" separation and tries to go as far as possible in hiding the low-level aspects of `M0`. Nevertheless should the programmer be able to manipulate the code produced by the PASCAL-M translator. For introducing this in a clean way, we provide messenger type declarations. A *messenger type declaration* is introduced by the new keyword `msgrtype` that has the structure of a parameterless procedure definition. Messenger types defined this way can not be called: instead, they define a string constant.

```
PROGRAM demo;
  CONST msg = 'a message';

  MSGRTYPE a;
  BEGIN ... END;

VAR s: string;
BEGIN { demo }
```

```
  s := msg;
  s := a;
  s := makemsgr(newkey, a, msg)
END;
```

The messenger type declaration for `a` results in a string constant that can be accessed like any other constant string value (e.g., `msg`). The actual messenger *instance* is created by the `makemsgr` function defined as

```
FUNCTION makemsgr(startqueue: key; mtype: string; mdata: string): string;
```

This function produces a messenger packet by including a 64-bit key that selects the process queue in which the messenger should be started as well as a data field which can be used to parametrize the messenger's code.

3.1.1 New Scoping Rules For Messenger Type Declarations

Because messengers are selfcontained programs shipped to remote locations, they may not contain references to variables or procedures defined outside the messenger's scope. The PTOM translator enforces this by allowing references to external type and constant definitions only: Variables, procedures and functions which are in a messenger's surrounding scope may not be referenced from within the messenger. Figure 3 depicts this new scoping rule that departs from standard PASCAL: a

```
PROGRAM a;
CONST b = 123;
TYPE c = char;
  PROCEDURE d;
  BEGIN ... END;
  MSGRTYPE e;
  VAR f: integer;
    PROCEDURE g;
    BEGIN ... END;
    MSGRTYPE h;
    BEGIN ... END;
  BEGIN { msgrtype e }
  END;
BEGIN { program a }
END.
```

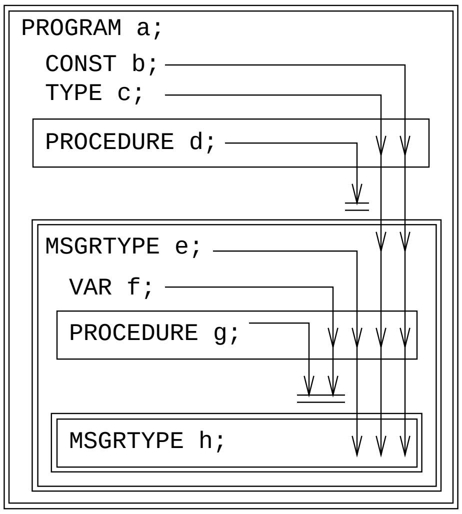

Figure 3: Visibility of names in messenger type declarations.

messenger type declarations "grounds" the usual nesting hierarchy of procedures and functions i.e., starts a new name space that inherits from its environment only constants and type definitions (including messenger types). Figure 4 shows this in text form for the example of figure 3: the table entries indicate whether a given name (rows) is visible for a given environment (columns) or not.

3.2 Other New Aspects of PASCAL-M

Our aim was to stay as close as possible to the original PASCAL language and not to make every feature of M∅ available to the PASCAL-M programmer. However, beside the messenger type declaration introduced we also had to extend the number of supported types to some degree. Character strings and string concatenation with

	PROGRAM a	PROCEDURE d	MSGRTYPE e	PROCEDURE g	MSGRTYPE h
b	*visible*	*visible*	*visible*	*visible*	*visible*
c	*visible*	*visible*	*visible*	*visible*	*visible*
d	*visible*	*visible*	–	–	–
e	*visible*	–	*visible*	*visible*	*visible*
f	–	–	*visible*	*visible*	–
g	–	–	*visible*	*visible*	–
h	–	–	*visible*	–	*visible*

Figure 4: Visibility of names for the example in figure 3.

`+` are in most PASCAL dialects (as well as JAVA) now state-of-the-art, so we do not consider this a novel extension. The `exit` statement for escaping loops and the `otherwise` clause for `case` statements are also rather standard today and are supported by PASCAL-M.

3.2.1 The `key` Type, Shared Memory Access and `inline` Procedures

The new simple data type `key` is a direct heritage from `M0` specifically introduced for the programming of messengers. In PASCAL-M keys are rather boring data instances: they only can be copied from one variable to another or be returned as a function's result. However, keys are essential for accessing the shared memory area, for handling process queues and for submitting messengers to other locations.

For example, the `globaldef` and `globalget` procedure enable to store (fetch) arbitrary data values in the platform's global dictionary. Under PASCAL-M entries are referenced by random keys. Only by knowing a key can other messenger access the corresponding data value. Using `globalundef`, entries can also be explicitly removed. With `globalknown`, one can check whether for a given key an entry already exists or not.

For implementation simplicity, we did not implement `M0`–specific procedures (like the above mentioned `global*` procedures) as PASCAL-M standard procedures. Instead, an `inline` mechanism is offered which enables to directly code a PASCAL-M function's `M0` instructions. For example, the `globaldef`[2] procedure is defined as:

```
PROCEDURE globaldef(k: key; a: any); INLINE = '_3 1R:';
```

The special type `any` allows for a generic handling (import/export from the shared memory area) of variables regardless fo their actual type. It is up to the programmer to ensure that data fetched from the shared memory area actually have the type of the receiving PASCAL-M variable. A possible extension would be to offer a `IS` predicate similar to Oberon (or `isinstanceof` in JAVA) by which a runtime type check can be explicitly enforced.

A collection of the most important "`M0` bindings" has been developed and consists of a set of "inlined" procedure declarations. This small `M0`–library that can be included in all PASCAL-M source code is shown in Appendix B. Although it is not part of the PASCAL-M language specification proper, its use is essential for programming

[2] Note that – according to the semantics of the shared memory area at the level of `M0` – the value stored by this procedure is a *copy* of the local variable instance. If later on the program needs to modify the *shared* data, it should reload the instance with a call to `globalget`.

messengers.

3.2.2 Summary of Main Differences to Standard PASCAL

Figure 5 summarizes the main differences between standard PASCAL, Wirth's small PASCAL-S (which was the base of the PTOM translator) and our messenger–specific PASCAL-M language. PASCAL-M features some relaxed syntactic rules aligned with

PASCAL	PASCAL-S	PASCAL-M
`real` variables	`real` variables	*not supported*
`alfa` for character strings	*not supported*	`string` type
pointer types	*not supported*	pointer types
variant records	*not supported*	*not supported*
set types	*not supported*	*not supported*
`forward` declaration	*not supported*	`forward` declaration
file IO	file IO	*not supported*
–	–	messenger type
–	–	`key` type
–	–	`any` type
–	–	`exit` for loops
–	–	`otherwise`
–	–	`inline` functions

Figure 5: Main differences between PASCAL, PASCAL-S and PASCAL-M

modern dialects of PASCAL (e.g., [10]): declaration blocks for variables, types and procedures can be placed in any order. Another syntactic departure of PASCAL-M regards the unit of compilation: PTOM can translate single functions (without the `PROGRAM` ... `END` structure), which is useful for writing small `MØ` procedures (for an example see section 4.4.1 and figure 13). The main difference is the size of generated code: without the `PROGRAM` ... `END` structure, the translator does not insert the PASCAL-M *runtime library* into the code stream. Note however, that the code strings generated *for messenger types* always contain the runtime library. The PASCAL-M runtime library should not be confounded with the above mentioned `MØ`–library: more on the runtime library can be found in section 4.3 and 4.4.1.

3.3 Possible Future Extensions of PASCAL-M

As we continue to express messengers with the new PASCAL-M language, we will certainly encounter other language features that would be nice to have. Some of them that we already know about are listed below, but we did not implement them yet:

- *Parametrized messengers:* parameters to messenger types can only be passed in a single string variable (see the `makemsgr` procedure) and implicitly in the new messenger's key for the starting queue. This requires the programmer to format (serialize) multiple parameters into a single parameter string and – inside the messenger – to decode (deserialize) the parameters. It would be elegant to have parametrized messenger declarations and letting the compiler insert the corresponding encoding and decoding routines directly into the messenger type's code string.

- *Conformant arrays* and the possibility to use them in formal parameter lists would be simple to implement in PTOM because `MØ` already implements high-level arrays with implicit length information.

- *Runtime type check:* as was mentioned above, it would be helpful to check a variable's type at runtime in case this variable was copied from the platform's shared memory. A `IS` predicate like the one found in Oberon [20] would do it.

- Because the PTOM translator knows the stack layout of a messenger (expressed in PASCAL-M), one could envisage the implementation of a `go` procedure like the one offered to mobile agents in TELESCRIPT [7]. The `go` procedure would have to "freeze" the complete state of the messenger (stack and execution state), encode it in some suitable form, transfer it to some remote platform and reinstantiate the messenger there. In fact, messengers would be used to implement higher level agents. It may be useful to envisage a new declaration header (such as e.g., `agenttype`) for messenger types that request explicit execution stack handling.

4 Internals of the PTOM Translator

The PTOM translator was not built form scratch. Instead, Wirth's well–known "small PASCAL" compiler was used as a starting point. This also means that PTOM is written in PASCAL. The PTOM translator currently compiles under Turbo Pascal 5 as well as with Sun Pascal 2.1. Major modifications to PASCAL-S were made in the following domains:

- The code interpreter that is part of PASCAL-S was removed, PTOM generates `MØ` code.

- File IO is not supported by PASCAL-M, all related procedures and types were removed from PASCAL-S.

- No efforts are made to recover a syntax error of the program under translation, thus the `error()` routine eventually calls `halt`.

- Instead of using calls to the read procedure, PTOM works on a full source code string (memory is cheap, mobile agents are small). Like PASCAL-S, we send code first to memory and dump it later.

- Adjustments were made to the set of supported data types and constructors: reals and associated data transfer procedures were removed, the pointers and enumerated types were (re-) implemented like in full PASCAL, "strings" and "keys" were added as new builtin types.

In the following paragraphs we discuss several of these points in more detail.

4.1 Code Generation Strategies

One of the interesting aspect of the PTOM translator is that code must be generated for a machine that provides high-level access to data objects only. Ordinarily, compilers for straight machine code have to handle memory allocation (a) for the generated code (including fixups), (b) for the execution stack of the running program and (c) for constants like strings and other static data. Translating for POSTSCRIPT–like target code results in significant simplifications.

4.1.1 Example 1: the `if-else` statement

Consider for example the code to be generated for an `if-else` statement with the boolean expression `expr` and the two code branches consisting of statements `stmt1` and `stmt2` (figure 6). In the case of assembler code, label L1 is required for jumping to

PASCAL	Assembler	POSTSCRIPT	M∅

`IF expr THEN`
`  stmt1`
`ELSE`
`  stmt2`
	`  expr`
	`  jz L1`
	`  stmt1`
	`  jmp L2`
	`L1:`
	`  stmt2`
	`L2:`
		`expr`
		`{ stmt1 }`
		`{ stmt2 }`
		`ifelse`
			`expr`
			`{ stmt1 }`
			`{ stmt2 }`
			`?`

Figure 6: Different implementations of an `if ... else` statement.

the second branch while label L2 points to the code where execution continues after the `if-else` statement. The exact location of L1 and L2 is not known at the moment when corresponding jump instructions must be emitted because at this point of time one does not know the exact code size of the statements `stmt1` and `stmt2`. A fixup operation is required that patches into already generated code the label addresses as soon as they become available (PASCAL-S does it this way). (Another possibility would be to assign index values to all labels (e.g., 1 and 2) and to set up a jump table where the addresses of L1 and L2 will be put. The jump instructions would then include a level of indirection, e.g., `jz jtab[1]` instead of `jz L1`). Looking at the translator code, this results in the simplified program structure of figure 7. In the case of `M∅` (or POSTSCRIPT), no fixup addresses are required: POSTSCRIPT's `ifelse` operator (`?` in `M∅`) expects two procedures on the operand stack from which one is chosen according to the boolean value (which also on the operand stack). In fact, it is the parser that examines the `{` and `}` symbols, grouping together the branches into two procedure objects and pushing them on the operand stack. The grouping is done once when the program is read in by the interpreter: this means that the execution of an `if-else` statement involves pushing the two (references to) procedure objects but is not slowed down by the grouping activity performed at parsing time. Note that `M∅` has no `if` operator (only an `ifelse` operator written as `?`) and therefore we always generate a second possibly empty procedure although there may be no `else` clause in the PASCAL program.

Compiler for Assembler code:	Translator for M∅ code:

<pre>if sy <> ifsy then error();
insymbol; expression;
if sy <> thensy then error();
insymbol; emit('jz XX');
l0 := location of XX;
statement;
if sy = elsesy then begin
  emit('jmp YY');
  l1 := location of YY;
  fixup(l0, current-code-position);
  insymbol; statement;
  fixup(l1, current-code-position)
end else
  fixup(l0, current-code-position);</pre>
	<pre>if sy <> ifsy then error();
	insymbol; expression;
	if sy <> thensy then error();
	insymbol; emit('{');
	statement;
	emit('}{');
	if sy = elsesy then begin
	  insymbol; statement
	end;
	emit('}?');</pre>

Figure 7: Generating code for an `if ... else` statement.

4.1.2 Example 2: Procedure Calls And Variable Access

`M∅` provides an operand stack where arbitrary data values can by pushed and popped. While it is easy to access the top element as well as elements at a given distance relative to the stack's top, it is difficult to address elements by an *absolute* index value. This renders techniques for accessing a procedure's local variables via a "frame" or "base" pointer infeasible. Keeping track of all arguments pushed and popped during a program's execution is error prone, wherefore we chose to use *dictionaries* with explicitly named variable entries for implementing a procedure activation frame. The prologue of a procedure then has the form `{dict begin` (in `M∅`, this becomes `{D(`), immediately followed by the instructions to pop off the arguments passed to that function and assigning them to variables in the newly created dictionary. The epilogue is simply a `end}` (`)}` in `M∅`) which automatically discards all local variables. Accessing local variables then becomes very easy, as shows the example in figure 8. In this example, the `_r` variable is inserted by the translator: we use this name by convention for a function's return value. Just before doing the epilogue, the content of this variable is pushed on the operand stack, leaving it there for the calling procedure.

Using POSTSCRIPT's dictionary stack for the activation frames has the advantage that name resolution (and thus locating the variable in memory) is carried out by the POSTSCRIPT interpreter and has not to be carried out by the translator. Thus, there is no need anymore for static and dynamic links through activation frames, although the translator still checks whether there is a scope wherein a given variable name is defined. More precisely, the access to a variable in some outer scope is programmed as shown in figure 9. The `find` operator (`F` in `M∅`) returns the topmost dictionary where the given name is defined and a boolean variable telling if a dictionary was found or not. Because the translator already checked that the variable is defined somewhere, we can always disregard this boolean value. Then we push the name of

PASCAL	POSTSCRIPT	M∅ (40 Bytes)
`function f(i: integer): integer;`	`/f {dict begin`	`.'f{D(`
	`/i exch def`	`.X'iX:`
`var s: integer;`	`/s 0 def`	`.'s0:`
`begin`		
`  s := i + 1;`	`/s i 1 add def`	`.'s i1+:`
`  f := s * s`	`/_r s s mul def`	`.'_r s s*:`
`end;`	`_r end} def`	`_r)}:`

Figure 8: Using a dictionary as an activation frame.

PASCAL	POSTSCRIPT	M∅ (42 Bytes)
`procedure a;`	`/a {dict begin`	`.'a{D(`
`var b: integer;`	`/b 0 def`	`.'b0:`
`  procedure c;`	`/c {dict begin`	`.'c{D(`
`  begin`		
`    b:= b + 123`	`  /b find pop /b b 123 add put`	`  'bFP'b b123+:`
`  end;`	`  end} def`	`  )}:`
`begin`		
`  b := 2;`	`/b 2 def`	`.'b2:`
`  c`	`c`	`c`
`end;`	`end} def`	`)}:`

Figure 9: Accessing a variable in a outer scope.

the variable to assign to, evaluate the right–hand side of the expression and use the `put` operator to store the value at the right place. Note that for accessing `b`'s value from within procedure `c` we do not have to care about the exact level of nesting i.e., about which dictionary contains this name: the interpreter will find it for us by scanning the dictionary stack.

4.1.3 Example 3: Passing Arguments By Value And By Reference

Finally we show how the PTOM translator generates code for passing arguments by reference (PASCAL's `VAR` arguments). As was shown in the previous examples, arguments for a procedure are pushed on the operand stack before calling the procedure or function, which is straight forward for value arguments. This has to be modified in the case we want to pass a reference instead of a value. In the case of a `VAR` argument we pass a two-element array which contains the dictionary *where* the value, and the name *under which* the value can be found. Since in M∅ and POSTSCRIPT an array occupies just one entry on the operand stack, we can use the old push/pop mechanism for passing by-reference parameters too. Figure 10 shows an example of this. Note that in this example the variable `d` now contains an array. The loop

PASCAL	POSTSCRIPT	MØ (42 Bytes)

`procedure a;`
`var b: integer;`
`  procedure c(var d: integer);`
`  begin`
`    d := 99`
`  end;`
`begin`
`  c(b)`
`end;`
	`/a {dict begin`
	`/b 0 def`
	`/c {dict begin`
	`  /d exch def`
	`  d {} forall 99 put`
	`end} def`
	
	`[currentdict /b] c`
	`end} def`
		`.'a{D(`
		`.'b0:`
		`.'c{D(`
		`  .X'dX:`
		`  d{}L99:`
		`)}:`
		
		`[.'b]c`
		`)}:`

Figure 10: Passing by-reference parameters.

over this variable pushes the array's elements on the operand stack such that they are perfectly aligned for the `put` operator that comes after having evaluated the right–hand side of the assignment. Of course it is the translator that keeps track of whether a variable actually holds a value or whether it only contains the reference to it. Correspondingly, the translator generates different code for an assignment statement.

4.2 Data Mapping

Simple PASCAL data types like integers, booleans and character values are all mapped to MØ integers. The PTOM translator generates operations which assert that the corresponding ranges are respected. PASCAL-M string values are naturally represented by MØ strings, although the translator must insert index adjustment computations because in PASCAL-M strings are implicitly defined as `array[1..] of char` while in MØ all arrays start with an index of 0. String concatenation is also simple, because MØ already offers a concatenation operator. The translator just has to care about forcing true string copies after concatenation and substring operations as well as before assignments because the MØ interpreter avoids as far as possible to physically copy string fragments, which may lead to undesired side effects at the level of PASCAL.

PASCAL-M arrays are mapped to MØ arrays. The translator inserts for all element accesses the required index offset. Accesses to elements beyond the array's index range are trapped by the MØ interpreter. When a new array variable is instantiated, the translator has to generate special instructions to also instantiate every element according to the element's base type (in native code compilers one just allocates a free memory area e.g., on the stack). Finally note that multidimensional PASCAL arrays are mapped to true `array of arrays` in MØ.

For records, the most natural way of representing them is again the *dictionary* data structure: a field with name a and of type integer, for example, becomes a dictionary entry where the name a is associated to an integer value. As with the elements of a new array instance, it is necessary to define all entries of a record as soon as a new record instance is needed. Using dictionaries has the advantage that the PASCAL `with` statements is very easy to implement: it suffices to push the record dictionary on the dictionary stack. Of course, the translator must remember this when it assigns a new value to a variable that has local scope but for which the associated dictionary is not the topmost dictionary anymore (see section 4.1.2).

Assignments of record and array variables require a special treatment by the translator. A simple memory copy is not possible in POSTSCRIPT or `MØ`, and other than the simple types, `MØ` does not support the assignment of composite types. Strings, as we already mentioned, are therefore explicitly duplicated with the `C` operator. However, this operator works only for the topmost level of a composite object: for example, a duplicated dictionary is an independent new object, but if some of the entries refer to other composite objects, these objects were not copied but are now shared by the two copies of the dictionary. The PTOM translator inserts for record and array assignments as well as for value parameters a call to the `_d` procedure, which duplicates even nested data structures. This procedure is not part of `MØ` and has to be furnished by the program itself: in fact, the `_d` procedure is part of the PASCAL-M runtime library (see also section 4.3).

4.2.1 Dynamic Memory Allocation

The PTOM translator generates code that automatically allocates variables when a procedure or function is activated (called). A lazy allocation of these variables i.e., waiting until the variable is explicitly assigned to by the programmer, does not work because of the way we make use of dictionaries. In section 4.1.2 we showed how the `find` operator is used to locate the topmost variable instance of a given name in order to assign to it a value. Without explicit allocation, this approach would not work anymore.

In PASCAL is the programming of dynamic memory allocation linked to pointer types. By using the `new(ptrvar)` and `dispose(ptrvar)`, the programmer can request the (dynamic) allocation and destruction of a variable instance. The PTOM translator uses `MØ` `key`s for implementing pointers. A special heap dictionary (`_h`) contains all associations between pointers (the `key`) and their variable instances. Figure 11 gives an example how this works. Allocating a dynamic variable (`new`) consists in creating

PASCAL	MØ (49 Bytes) and comments
`var p: ^integer;`	`.'p0:         # automatic allocate of pointer`
`    i: integer;`	`.'i0:         # automatic allocate of integer`
`begin`	
`  new(p);`	`.'p;_h1I0::   # create a random ptr entry in _h`
`  p^ := 99;`	`_h p 99:      # assignment via ptr`
`  i := ^p;`	`.'i _h pG:    # dereference the ptr variable`
`  dispose(p);`	`_h pU         # undefine the ptr in _h`
`  p := nil`	`.'p0:         # nil is defined as 0`
`end;`	

Figure 11: An implementation of `new` and `dispose`.

a random key k (which becomes the pointer value stored in the pointer variable) and associating it to a newly created variable instance in the `_h` dictionary. Pointer dereferencing is a simple lookup of this key in the `_h` dictionary, assignment to the variable pointed to by a pointer also refers to the heap dictionary. The `dispose()` procedure is implemented as a simple `undefine` i.e., we remove the entry with key k from the heap dictionary. This has the nice property that stale pointers (pointer to freed memory zones) automatically generate a runtime error that is trapped by the `MØ` interpreter: because `dispose` removes the entry in `_h`, a subsequent `G` (get) due to the dereferencing of the stale pointer will fail. `Nil` is represented as the integer value 0. Currently there is no support for trapping *assignments* to a `nil` pointer (which would be simple to add). Attempts to access a variable via a `nil` pointer, however, are trapped as long as there is no entry for the value `0` in the heap dictionary (such an entry could appear when one assigns a value to a `nil` pointer).

4.2.2 Name Space Mapping

Classical compilers usually drop variable and procedure names once the parsing and code generation is completed (except for symbol tables for debugging purposes and external references for the linker phase). In fact, variable references are replaced by memory addresses. In our approach we keep the names and use them for all references to variables and functions. The PTOM translator just enforces that the characters used in the names are compatible with the lexical rules of `MØ`. The case to solve are digits as part of an identifier, because `MØ` limits multi-character identifiers to consist of lowercase characters `'a'...'z'` only. The PTOM translator replaces the digits `0`, `1`, etc. found in PASCAL identifiers by `_a`, `_b` etc. In order to avoid potential name clashes when the underscore character is inadvertently used at the PASCAL-M level, underscores are also escaped i.e., replaced by two underscores at the level of `MØ`.

4.3 The PTOM Runtime Library

Most of the time the PTOM translator generates plain `MØ` instructions without considering optimization like avoiding frequent repetition of more complex instruction sequences. However, there are some cases where factoring out common instruction sequences is adequate or even mandatory. For these procedures, which form a kind of small run–time library, the following names were defined (we also list other names used internally by the runtime):

- `_d` dup procedure: recursively duplicates composite objects like arrays or records.
- `_e`: reserved for a possible runtime error code (only present in the runtime library of a `program`).
- `_h` heap dictionary: usually an empty dictionary at start time, but this may also point to the global `MØ` heap.
- `_r` function's return value: this special name is used in functions as the "return variable" (using the function's own name is not possible because it is already used for referencing the function's procedural body).
- `_s` stringify procedure: turn a character (integer) value into a string of length 1.
- `_z`: contains miscellaneous data for gracefully halting a PASCAL-M program (only present in the runtime library of a `program`).

PASCAL-M code	generated M∅ code
`MSGRTYPE a;`	`...         # code of runtime-library`
	`            # would be placed here`
`  VAR b: string;`	`.'bOS:      # instantiate b (empty string)`
`  MSGRTYPE c;`	
`  BEGIN ???? END;`	
`  PROCEDURE d;`	`.'d{(...)}: # define procedure d`
`  BEGIN ... END;`	
`BEGIN`	
`  b := c`	`.'b"????":  # body of a (the assignment)`
`END;`	

Figure 12: Code for messenger types is not output at declaration time.

This runtime library is automatically produced by the PTOM translator. Because some of its entries are useful only for complete PASCAL-M programs, code strings of messenger types do not include all elements of the runtime library. The PTOM translator omits corresponding entries (`_e`, `_z`) according to the absence of a `program` keyword.

4.4 Code Output

The introduction of the `msgrtype` declarations leads to some unusual considerations about the time when code is actually output. The reason for this is that messenger types behave like string constants i.e., they are inlined into the code at the places where the types are referenced which is different from the place where they are declared. Between these places, a PASCAL-M compiler must store the messenger type string in internal memory.

In the PTOM translator, the generation of code that instantiates local variables of a procedure or messenger type is also delayed, but now for implementation reasons. The case to consider is the mix of messenger type declarations and the declarations of local variables. In PASCAL-M we have to generate explicit variable instantiation code, as was explained in section 4.1.2. Consider the code fragment and its translation in M∅ in figure 12. The goal is to have, after having processed the messenger type `a`, a single string in memory for `a`. In a first attempt one could try to emit code as soon as possible: when we see the `MSGRTYPE a;` header, we could emit the PASCAL-M library, when we see the `VAR b: string` line, we would generate the variable instantiation code immediately following the library code etc. However, when we encounter the declaration for the messenger type `c`, we cannot emit its code right now: it has to be inserted only when it is referenced in the body of messenger `a`. So we would have to store the code for `c` in another memory area. Therefore we delay writing code for messenger `a` until the start of its body so that we can use the linear memory area for first writing the code of messenger type `c`. When `a`'s body commences, we just have

PASCAL-M code	generated M∅ code

`PROGRAM a;`
`  VAR b: integer;`
`  PROCEDURE c;`
`    BEGIN ... END;`
`  PROCEDURE d;`
`    BEGIN ... END.`
`BEGIN`
`  ...`
`END.`
	`...         # code of runtime-library`
	`             # would be placed here`
	
	`.'b0:        # instantiate variable b`
	`.'c{(...)}: # define procedure c`
	`.'d{(...)}: # define procedure d`
	
	`...         # body of a`

`VAR b: integer;`
`PROCEDURE c;`
`  BEGIN ... END;`
`PROCEDURE d;`
`  BEGIN ... END.`
	`.'b0:        # instantiate variable b`
	`.'c{(...)}: # define procedure c`
	`.'d{(...)}: # define procedure d`

Figure 13: Suppressing the generation of the PASCAL-M runtime library.

to go through the translator's data structures in order to generate the instantiation code for variables (and the prologue if a were a procedure etc).

What we explained here for local messenger type declarations also applies to local procedure declarations: their code too is written into the memory area *before* the outer messenger type or procedure's body starts. This means that we duplicate the code of local procedures: a first time it is "temporarily" written when the declaration is encountered, and then we copy this code when the actual body of the outer procedure starts. It would be possible to reclaim the memory occupied by the "precipitately" emitted code of local procedures by shifting the finalized messenger type code over it, but this has not been implemented yet.

4.4.1 Two Different Translation Modes

The PTOM translator can be used to produce M∅ code for small procedures and functions expressed in PASCAL-M *without* also emitting the PASCAL-M library. This is useful when one assembles a set of PTOM generated procedures by hand and one does not want to manually remove the runtime library included by default. This behavior is selected by simply omitting the usual `PROGRAM name;` header line (no main body is then allowed, however). Thus, both PASCAL-M code fragments of figure 13 are valid compilation units.

4.5 Example of a Messenger written in PASCAL-M

The following example shows how easy it is to write a concurrent messenger application, although the problem it solves is not very relevant. The bubble sort algorithm was "parallelized" such that several messengers can compare adjacent element of an array concurrently. The PASCAL-M sources of the `bsort procedure` are given in appendix A, figure 14 shows the possible execution paths of the messengers that are created by this procedure.

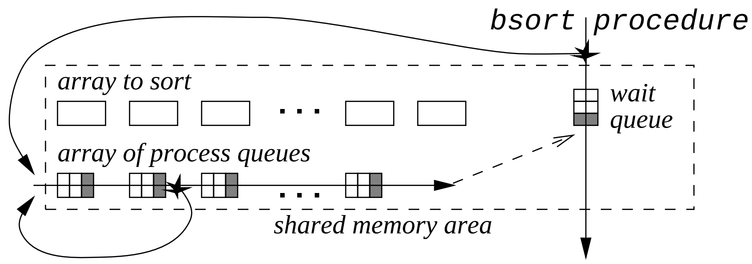

Figure 14: The execution paths of the `bsort` messengers.

The main flow of execution is the `bsort` procedure. First, it stores the array to sort in shared memory for letting the concurrent "worker messengers" access it. After some more initialization work (setting up an array of process queue for "concurrency control"), it starts a first "sweeper" messenger and blocks in a separate process queue waiting for the signal that sorting is done. The sweeper starts comparing the elements from index position 1. When it encounters two elements in wrong sequence, they are swapped and the sweeper messenger moves to the next array position. The first time a sweeper swaps some elements, it creates another sweeper messenger that starts all over again (thus, the number of concurrent messengers depends on the data in the array). Eventually, there will be a messenger that sweeps the full array without encountering any element pairs to swap: this final sweeper unblocks the waiting `bsort` messenger thread. Note that all sweeper messengers proceed in an array of process queues in order to prevent race conditions on array entries.

Because all sweeper messengers have the same task, a messenger instance is created only once with `makemsgr` and stored in the shared memory area. When a sweeper has to start another sweeper process, it can use this pre-assembled messenger packet. The PASCAL-M code for the `bsort` procedure in appendix A furthermore shows that the sweeper messenger is created without explicit passing of arguments (i.e., the data argument is an empty string). Nevertheless requires each sweeper messenger a key under which it can access the data in the shared memory area. This key is *implicitly* passed in the messenger's start queue (first parameter of `makemsgr`): this explains why the sweeper messenger gets its current process queue and uses this key for accessing the record in the shared memory area. It is the same queue that the `bsort` procedure uses for storing the data record in the shared memory area.

5 Discussion

The PTOM translator does not break any speed records nor does it attempt to produce perfect `MØ` code. There are several areas where one could enhance it:

- PTOM does no optimization: unused variables and procedures, dead code etc. are all ignored i.e., they nevertheless generate code.

- The deeper a program makes recursive calls, the more there will be a runtime overhead in the `MØ` interpreter. This has to do with the way we implement activation frames by pushing local dictionaries on the dictionary stack. This means that the `M0` interpreter must traverse the complete stack every time it has to resolve the name even of a *builtin* `M0` operator. Therefore, one should use the bind operator and replace `M0` operator names by the dereferenced operators themselves.

- This optimization could be extended to also bind names of user-defined PASCAL-M procedures (most variables, of course, must remain unbound because of the dynamic/static link theme).

- A simple peephole optimizer would eliminate most of the idempotent instruction sequences (e.g., duplifying a string result twice, consecutive initialization and assignment etc.).

More of conceptual nature is the observation that the dichotomy between the messenger's local and the platform's shared memory is difficult to express in a natural way in PASCAL-M. Our view is that one should not mask or hide the explicit import and export of data (currently expressed by the `globaldef` and `globalget` procedures).

It would be nice to demonstrate PTOM's capacity by letting the translator translate itself. However, for this the `set` type would have to be implemented.

Finally we mention that the current PTOM translator provides virtually no debugging aids. Using the `H` operator of `M0` (`stopped` in POSTSCRIPT), it would be easy to produce code that can catch runtime errors and, by including line number information, that could indicate where the error happened. Such support functionality could, for example, be controlled by Wirth's compiler options convention (`{$D+ }`).

5.1 Related Work

`M0` can be classified in several ways. First, it was conceived as a language for human programmers providing all necessary functionality of a *high-level language* (no memory addresses, only typed data values etc). At the same time is `M0` used as the machine independent *representation of executable programs* (messengers). Finally, from the viewpoint of the PTOM translator, `M0` becomes an *intermediate language*. *Native code* is not part of `M0`, but the `M0` interpreter has been used to support the execution of native code under the control of messengers [16]. These four terms span the field of work that is related to messengers and mobile code in general:

- Most prominent, JAVA is an object oriented language for writing applets i.e., programs that are downloaded to and executed in the user's WWW–browser [8]. The `javac` compiler is used for turning high-level JAVA code into "byte codes" for a stack-based virtual machine (JAVA-VM) i.e., an interpreter. Programs at the byte code level have not the same flexibility as POSTSCRIPT or `M0` code: metaprogramming (code generation at runtime) heavily depends on the runtime environment, i.e., availability of compiler access at runtime (see e.g., [14]). The JAVA language and interpreter has been used to implement a mobile agent systems [15]: mobility is handled in a transparent way not influencable by the agent programmer, code and agent state travel in independent transport channels.

- TELESCRIPT, which predates JAVA as a language for mobile agents, has a similar approach with two languages (one high-level and one FORTH-like low-level TELESCRIPT), but was designed right from the start to be used for mobile code [7]. There is a high-level `go` operator that triggers an agent's displacement: even the program's *current point* of execution is transferred.

- TACOMA is a mobile agent system that is based on the scripting language `Tcl`, so there is no compiling or translating involved [12].
- The transfer of the state of a stack-based program is examined in the manual for the "Transportable PostScript" (Tps) language [9].
- For an overview and classification of still other languages for mobile code we refer to [3].

The use of intermediate code in compilers has a long tradition that, with the advent of mobile code, is suddenly boosted. The UCSD-PASCAL produced p-codes for a easily portable interpreter [2]. M-codes are used as intermediate representation of programs written in *Clarity C++* [13], byte codes were used for efficient SMALLTALK interpretation [4]. Intermediate code representations are also studied for on-the-fly code generation [6]. Compiling stack-based languages may in the future become an important issue for mobile code: for a study on FORTH see [5].

We are not aware of any translator/compiler that turns PASCAL code into POSTSCRIPT or any other high-level stack-based language.

Acknowledgments

P.-A. Queloz at the University of Geneva provided valuable comments on a first draft of this report.

References

[1] Adobe Systems Incorporated, editor. *PostScript Language: Reference Manual.* Addison-Wesley, fifteenth edition, 1990.

[2] Kenneth L. Bowles. *Microcomputer: problem solving using PASCAL.* Springer, 1977.

[3] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing mobile code languages. In *Proceedings of the 2nd Workshop on Mobile Object Systems,* ECOOP *conference 1996*, July 1996.

[4] L. P. Deutsch and A. Schiffmann. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, 1984.

[5] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technische Universität Wien, April 1996.

[6] Michael Franz. Technological steps towards a software component industry. In *Programming Languages and System Architecture*, March 1994.

[7] General Magic Inc. *Telescript Language Reference*, 1995.

[8] J. Gosling and H. McGilton. The JAVA language environment: A white paper. Technical report, Sun Microsystems, 1995.
URL `ftp://ftp.javasoft.com/docs/whitepaper.A4.ps.tar.Z`.

[9] Dennis Heimbigner. *The Tps Reference Manual, Version 2.1a.* University of Colorado Arcadia Project, March 1995. Arcadia Technical Report `CU-ARCADIA-104-05`.

[10] Tomy Hetherington. An introduction to the extended pascal language. *ACM SIGPLAN notices*, 28(11):42–51, November 1993.

[11] Kathleen Jensen and Niklaus Wirth. PASCAL *User Manual and Report.* Springer, 2nd edition, 1978.

[12] D. Johansen et. al. An introduction to the TACOMA distributed system. Technical report, Tromso University, 1995. Computer Science Technical Report 95-23.

[13] B. T. Lewis, L. Peter Deutsch, and T. C. Goldstein. Clarity MCode: A retargetable intermediate representation for compilation. *ACM SIGPLAN notices*, 30(3):119–128, March 1995.

[14] Netscape. The JAVA runtime interface, version 0.5.
URL `http://home.netscape.com/eng/jri/`, May 1996.

[15] M. Strasser, J. Baumann, and F. Hohl. MOLE: A JAVA based mobile agent system. In *Proceedings of the 2nd Workshop on Mobile Object Systems,* ECOOP *conference 1996*, July 1995.

[16] Chr. F. Tschudin, G. Di Marzo, Muhugusa Murhimanya, and Jürgen Harms. *Messenger–Based Operating Systems*, July 1994. Technical report 90 (Cahier du CUI).

[17] Christian F. Tschudin. *On the Structuring of Computer Communications.* PhD thesis, Université de Genève, 1993. Thèse No 2632.

[18] Christian F. Tschudin. *An Introduction to the* `MØ` *Messenger Language.* Technical Report 86, Centre Universitaire d'Informatique, May 1994. Cahier du CUI.

[19] Christian F. Tschudin. *M0 - a messenger execution environment.* Usenet newsgroup `comp.sources.unix`, Vol 28, Issue 51–62, June 1994.

[20] Niklaus Wirth. From Modula to Oberon – The Programming Language Oberon. Technical report, ETH Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Computersysteme, November 1990. Number 143.

A A Messenger-Based Parallel Bubble Sort in PASCAL-M

See section 4.5 for a description of the algorithm and figure 14 for a pictorial view of the execution path of the concurrent "sweeper" messengers.

```
CONST n = 10;
TYPE  eltype = integer;
      artype = ARRAY[1..n] OF eltype;
      rec = RECORD             { record for shared memory data: }
        a: artype;             {   array to sort }
        s: string;             {   sweeper messenger code }
        q: ARRAY[2..n] of key; {   array of process queues }
        w: key                 {   wait queue for bsort procedure }
      END;

{ The MO library (declarations for inlined functions and
  procedures) would go here: see the Appendix on next page. }

PROCEDURE bsort(VAR arr: artype);
VAR i: integer;
    k: key;
    r: rec;

  MSGRTYPE sweeper;
  VAR b: boolean;    { true if did not swap elements yet }
      i: integer;
      r: rec;
      x: eltype;
  BEGIN { sweeper }
    globalget(currentqueue, r); b := true;
    FOR i := 1 TO n-1 DO WITH r DO BEGIN
      queueenter(q[i+1]);
      IF a[i] > a[i+1] THEN BEGIN
        x := a[i]; a[i] := a[i+1]; a[i+1] := x;
        IF b THEN BEGIN
          localsubmit(s);
          b := false
        END
      END
    END;
    IF b THEN queuestop(r.w, false)
  END; { sweeper }

BEGIN { bsort }
  k := newkey;
  WITH r DO BEGIN
    w := newkey; FOR i := 2 TO n DO q[i] := newkey;
    a := arr; s := makemsgr(k, sweeper, '');
    globaldef(k, r); queuestop(w, true);
    localsubmit(s); queueenter(w)
  END;
  queueleave; globalget(k, r); arr := r.a;
  globalundef(k)
END; { bsort }
```

B The M∅ library of "inlined" procedures

The following declarations of functions and procedures form the M∅ library that every PASCAL-M program has to include if it wants to use the messenger specific functionality of the underlying M∅ interpreter (`submit` etc). If the PTOM translator is used just as a convenient way to produce M∅ code for ordinary (non-mobile code) applications, this library needs not to be included.

Note that the PTOM translator replaces all calls to functions and procedures of the M∅ library by the string constant given behind the `inline` keyword. If such an inlined function is used many many times in a program, it may be useful to add a private function that just calls the inlined library function. This is less efficient but results in only one copy of the inlined code to be included in the final M∅ code. Note also that the names of inlined functions do not appear in the output of the PTOM compiler while names of ordinary functions are copied "as is". If code size is a concern, you should use short names for procedures as well as variables.

```
{ M0 lib --------------------------------------------------------------- }

FUNCTION  code: string;                INLINE = '_cod';
FUNCTION  data: string;                INLINE = '_dat null={0S}{_dat}?';
FUNCTION  hostid: key;                 INLINE = '_hid';
FUNCTION  newkey: key;                 INLINE = ';';
FUNCTION  notkey(k: key): key;         INLINE = '~';

FUNCTION  globalknown(k: key): boolean; INLINE = '_XK';
PROCEDURE globaldef(k: key; a: any);    INLINE = '_3 1R:';
PROCEDURE globalget(k: key; var a: any); INLINE = 'X_XGX{}L3 2R:';
PROCEDURE globalundef(k: key);          INLINE = '_XU';

FUNCTION  currentqueue: key;            INLINE = ',';
PROCEDURE queueenter(k: key);           INLINE = 'E';
PROCEDURE queuetimeout(k: key; millisec: integer; var timeout: boolean);
                                        INLINE = '3 1R1000*@+EX{}L3 2R:';
PROCEDURE queueleave;                   INLINE = 'V';
PROCEDURE queuestop(k: key; stpd: boolean); INLINE = 'Q';

FUNCTION  makemsgr(k: key; mtype: string; mdata: string): string;
                                        INLINE = '[4 1R]_ctm';
PROCEDURE submit(k: key; msgr: string); INLINE = '$';
PROCEDURE localsubmit(msgr: string);    INLINE = 'nullX$';

FUNCTION  eval2int(s: string): integer; INLINE = '!0IY''integer={}{Z}?';
FUNCTION  eval2str(s: string): string;  INLINE = '!0IY''string={}{Z}?';
FUNCTION  eval2key(s: string): key;     INLINE = '!0IY''key={}{Z}?';

PROCEDURE log(s: string);               INLINE = '"\x0a"+{_log}HP';
FUNCTION  bool2str(i: integer): string; INLINE = '{"false"}{"true"}?';
FUNCTION  int2str(i: integer): string;  INLINE = '0I0<{N_cte"-"X+}{_cte}?';
FUNCTION  key2str(k: key): string;      INLINE = '_cte[0 18]G';
FUNCTION  random: integer;              INLINE = ';_cts[0 4]G_cti';

{ end of M0 lib -------------------------------------------------------- }
```