

5.0 References

- [Abr85] S. ABRAMSKI, R. SYKES. SECD-m: a Virtual Machine for Applicative Programming, In *Functional Programming Languages and Computer Architecture, Nancy, France*, LNCS 201, Springer Verlag, september 1985.
- [Ber85] B. BERTHOMIEU. Le langage LCS, une implantation expérimentale de CCS, In *Premier colloque C-Cube, Angoulême, France*, A. Arnold Ed., september 1985.
- [Ber88] B. BERTHOMIEU. LCS, une implantation de CCS, In *Troisième colloque C-Cube, Angoulême, France*, A. Arnold Ed., december 1988.
- [Car84.1] L. CARDELLI. *The Amber Machine*, Technical Report, AT&T Bell Laboratories, 1984.
- [Car84.2] L. CARDELLI. Compiling a Functional Language. In *ACM Symposium on Lisp and Functional Programming, Austin, Texas*, august 1984.
- [Car84.3] L. CARDELLI. An Implementation Model for Rendezvous Communication. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA*, LNCS 197, Springer Verlag, july 1984.
- [Dam82] L. DAMAS, R. MILNER. Principal type-schemes for functional programs. ACM 1982.
- [Gor79] M. GORDON, R. MILNER, and C. WADSWORTH. *Edinburgh LCF*. LNCS 78, Springer Verlag, 1979.
- [Hol83] S. HOLMSTROM. PFL: A Functional Language for Parallel Programming. In *Declarative Programming Workshop*, Programming Methodology Group, Chalmers University of Technology, University of Goteborg, Sweden, 1983.
- [Inr84] INRIA. The ML Handbook. Projet FORMEL, INRIA, 1984.
- [Lan64] P.J. LANDIN. The Mechanical Evaluation of Expressions. *Computer Journal*, 1964.
- [Mil78] R. MILNER. A Theory of Type Polymorphism in Programming Languages. *Journal of Computer and System Science*, 17, 1978.
- [Mil80] R. MILNER. *A Calculus of Communicating Systems*, LNCS 92, Springer Verlag, 1980.
- [Mil80] R. MILNER et al. Standard ML Proposal. Polymorphism, the ML/LCF/Hope newsletter, 1985.
- [Mit86] K. MITCHELL. *Implementations of Process Synchronisation and their Analysis*. PhD thesis, University of Edimburgh, july 1986.
- [Wan87] M. WAND. Complete Type Inference for Simple Objects. In *Symposium on Logic in Computer Science, Ithaca, New York*, june 1987.

User interfaces

The user interfaces are entirely written in LCS. The top-level function is implemented in several passes, taking care of parsing, typechecking, (abstract) code generation, execution (by a change of level of execution in the machine, and return), user environment updating and printing of results. At machine level, the user interface is seen as a regular LCS process.

The machine can be rebooted on any user provided function. The user may save a core image of the current environment or restore a previously saved image. For portability, core images are ascii files and their format is independent from the physical machine on which LCS runs.

The LCS executive interface needs all the features of the LCS machine, the simulator interface only needs ML capabilities as behavior constructions are interpreted, rather than executed, in that environment. The LCS executive interface takes about 6000 lines of ML, producing a core image of approximately 145 Kb.

4.0 Conclusions

As a language design experiment, and beside its extensions of CCS, LCS brings a new method for typing behaviors that, certainly, could be used for typing other language constructs (records for instance, as in [Wan87]), though we did not attempt yet to apply this idea in other contexts.

As a language implementation experiment, LCS brings its elaborated implementation of an higher order language with all capabilities of CCS, plus computed communication ports and support of side-effects. The experiment leads to the design of a virtual machine with several unique features such as strong processes at allocation level, automatic memory allocation and reclamation combined with copy-on-write for process stores, handling of unguarded non-determinism, true time-slicing, automatic orphan processes elimination (process pruning), inter process communication by rendez-vous, etc.

Finally, as a programming tool, LCS allows to comfortably experiment the parallel programming concepts introduced by CCS, and to gain experience in using such formalisms. It is also a valuable teaching tool for modern parallel programming concepts.

The Format layer stack serves as the root of the garbage collection routine. The routine implemented is a stop-and-copy, one-pass, two-spaces, compacting algorithm. Collection proceeds "per process", collecting one process after the other; collection ends when all processes have been collected. Collecting a process consists of recursively copying in the new space all cells referenced from its frame (or from the stack if collection was initiated by that process), incrementally rebuilding its store. The copy preserves both the sharing (by processes) of cells encoding non-updatable data and the sharing of store pages due to the copy-on-write store management.

Application layer

The Data sub-layer implements the data structures manipulated by the abstract machine into their Format level representation. These are either LCS data structures (e.g. pairs, closures, code, etc.), or data structures which are private to the machine (e.g. positions, lines, processes, process queues, etc.). Several distinct objects at Data level may have identical representation at Format level; as the Amber machine, the LCS machine is not type safe by itself.

The Abstract machine is implemented with 8 registers and the stack. These registers are E (environment, a linked list), C (the cell packing the code), PC (program counter), R (resumption register, a stack height), P (process positions, encoded as pairs of values), L (process lines, encoded as linked lists), G (positions accumulator, encoded as P) and Rp (ready processes, encoded as a queue of processes); the stack provided by the Format layer encodes both the S and D abstract registers. There are about 40 basic instructions, plus a number of derived instructions representing frequent sequences of the former, and about 70 primitives (logic, arithmetic, input/output, etc.).

Registers Rp and G are shared by all processes; although register L is local to processes, the queues of processes it associates with ports are shared by all processes. When a process is suspended, following an unsatisfied communication offer, then its stack is empty and needs not be part of the context of the process. Upon I/O suspensions, process contexts are constituted of registers E, C, P and L (and, transparently, of the store allocated to the process at format level).

The scheduling discipline is basically round-robin, but, randomly, the scheduler skips the first non obsolete process in Rp and places it back at the end of the queue; this breaks repetitive scheduling sequences and improves fairness. This strategy is further combined with a time-slicing so that all ready processes can make progress, even if some of them perform no communications (e.g. the top-level process). When interrupted by time-slicing, and conversely to the above case, the stack of a process is generally not empty. When this (infrequent) case occurs, the stack of the process is first saved in the heap, as a vector, with the contents of registers R, PC and C pushed on it; a stack resumption code is then forced onto the C code register, and the patched process is suspended as a regular I/O suspension.

Unguarded sums are supported, but with some restrictions, in the current implementation. Process positions (remember they encode a concurrence relation) allow, as encoded, an embedding of at most 31 process sums. Process pruning is implemented at operational level, it occurs when the bound on ambiguity depth has been reached. All non-obsolete processes are then scanned, their position information factorized by that accumulated in the G register, their lines pruned for the obsolete processes they contain, and the G register is reset.

The LCS machine has been implemented as a carefully optimized bytecode interpreter written in C (about 6000 lines). It runs under a variety of UNIX systems (BSD and System V) and under APPLE MacIntosh. Several variants of the virtual machine are being investigated, including distributed implementations on multiprocessor machines and on a network of machines.

register, a machine which is concurrent with the local position and offering a matching communication (on the same port, in opposite direction). If found, the partner is removed from the queue it was found into, otherwise the requester leaves its address in the adequate queue in L, and pauses. If a partner has been selected, the sending machine keeps in its context a reference to the receiver and computes the value to be sent. When computed, the message is pushed onto the environment of the receiver, and the receiver is resumed.

If infinitely many processors were available, then the view of the LCS machine as a collection of SECDR machines, with the above features added, would be enough. But, as this is not the case, the machine is enriched of a register: Rp (ready processes), that holds a queue of processes (i.e. of machine states), waiting for allocation of a machine.

The following rules are adopted for machine allocation and scheduling. Register Rp is managed as a FIFO queue. Upon an unsatisfied communication offer, or upon termination, the scheduler will be called to allocate the next ready process to the freed machine. In a rendez-vous conclusion, the receiver is resumed, and the sender is suspended. One of the processes created upon execution of one of the LCS composition combinators will be placed in the Rp queue of processes, while the other will continue execution.

Finally, process pruning (removal of obsolete process summands) is made periodically, as its cost would be very high if realized at each move. An additional register, G, is provided, shared by all machines, that accumulates, between two prunings, the positions of all processes that made a move. The concurrence test between processes is augmented by a "non-obsolescence" test, implemented using that G register. A process is non-obsolete if it is concurrent with all the processes that performed a move since the last pruning (their positions are found in G). Process pruning consists of removing all the obsolete processes from the system (found in shared Rp and local L registers), and of resetting the G register to the empty set.

There is a machine instruction for each behavior construct, plus one instruction for rendez-vous conclusion, one for agent start at top-level.

Implementation, the LCS virtual machine

The first two layers of the implementation constitute the Virtual Machine. The first of these layers, the Format layer, offers memory management and simple process management services; the second layer implements the abstract machine on this ground, it is constituted of the Data and Operational sub-layers.

Format layer

The Format layer includes a memory allocator, a process allocator with store handling routines, a garbage collector, a dumper / loader for core images and an adressable stack.

Non assignable LCS and machine data are kept in a heap, shared by all processes, either as vector cells (holding constants and/or cell references), or as packet cells (holding byte sequences). Assignable data (e.g. LCS references) are kept in specific structures called Stores. Location spaces for processes are paginated, page tables and data pages are kept in the heap, as cells of a third kind (pages). A process is the association of a context (a vector) with a store, process frames constitute the fourth and last kind of heap cells. The process creation routines (light-fork, strong-fork, load and switch), together with the store management primitives (content, assign, newref), implement a lazy copy of process stores following a "copy-on-write" strategy.

LCS. The simulator allows to expand, under full control of the user, one agent at a time; the CCS expansion theorem [Mil80] constitutes the expansion rule there.

3.0 The Implementations

Architecture of implementations

Among requirements for LCS were very fast strong processes creation, combined with automatic memory allocation and reclamation. LCS processes are not implemented as host's operating system processes; process management is taken care of by the LCS virtual machine.

Other features of the machine include a particular treatment of non determinism (unguarded sums are supported), interprocess communication and synchronization by rendez-vous, a both communication bound and time bound scheduling, automatic orphan process elimination, both strong processes and light processes (threads), copy-on-write memory allocation for process stores, etc.

Following a now classical method (e.g. [Car84.1]), LCS implementations are structured in three layers. A first layer handles automatic memory allocation and reclamation and, here, process creation; a second layer implements an abstract machine, with its registers and instructions; the third layer is constituted of the user interface, including an incremental compiler, and itself written in the language being implemented.

The LCS abstract machine

The ML sub-machine of the LCS machine is essentially an SECD machine, enriched of a resumption register R for exception handling, customized for execution of ML, and with instructions implementing the change of level of execution and restart of the machine on a new application. This machine is called "SECDR machine" in the sequel.

The LCS abstract machine can be seen as a set of SECDR machines, associated with a concurrence relation involving these machines, and with a mechanism allowing these machine to communicate and synchronize. The system of machines progresses by moves of one or several machines, following a commitment or rendez-vous; by replacement of some machine by two others, following a composition; or by deletion of a machine. Machine moves are considered atomic, and include a machine pruning operation, that removes, at each move, from the system of machines, those that are not concurrent with the machine(s) performing the move.

The concurrence relation among machines is encoded by associating with every machine in the system a "position" information, kept in a local P register, which allow to decide, comparing their respective positions, if two machines are concurrent or not. When agent compositions are executed, the concurrence relation is incrementally extended to handle the newly created machines.

For inter-machine communication, each machine is provided with a "Lines" register L, organized as a linked list, that holds, for each port label reachable from this machine, a pair of queues of (references to) machines: those waiting for input on a port with that label, and those waiting for output on a port with that label; these queues are shared by all machines. Renamings (resp. hidings) will push copies of lines (resp. new lines) on the L register of the machine on which this construct is executed.

Rendez-vous is split into three steps: partner selection, computation of the message and passing of the message. This allows to make execution of several rendez-vous overlap. Partner selection is atomic, it consists of finding in the Lines

LCS behavior variables have similarities with Wand's row-variables introduced in [Wan87], in which they are used for typing records. Despite this, type inference here and in the mentioned reference have important differences, as the support of behavior types (the set of labels to which types are assigned) is not required to be a finite set here.

Types are assigned to all sub-expressions in a context that assigns a type to each identifier. Type assignment for ML expressions obey the rules explained in [Dam82][Mil78]; the typing rules for behaviors are summarized below:

- All occurrences of an identifier bound by abstraction or by input must receive, in their scope, the same type. Occurrences in their scope of locally, or top-level, declared identifiers receive as types generic instances of the types assigned in their declaration;
- `nil` and compositions (`/\`, `\/`, `//\`, `\\/`) are typed as occurrences of top-level bound identifiers with types `.` and `.` \rightarrow `.` \rightarrow `.`, respectively;
- In actions, the suffix agent must have a behavior type. For input and output actions, the type assigned to the label occurring in the action, if any, must be the same as the type assigned to this label in the type of the suffix agent;
- In a hiding, the type constraints assigned to the labels being hidden, if any, are relaxed in the result type. e.g. in `"agent A a = a {/p}"`, A has type `{p:'a}. \rightarrow {p:'b}`.
- In a renaming, the type of the agent involved must be such that each renamed label has the same type that the label in which it is renamed. The type constraints of renamed labels are then relaxed in the result type. e.g. in `"agent A a = a {q/p}"`, A has type `{p:'a, q:'a}. \rightarrow {p:'b, q:'a}`.

Semantics

Evaluation is applicative. All behavior constructs evaluate as closures. Closures capture the current environment, together with the (code generated for the) body of the function or behavior. The body of a behavior closure will be evaluated upon an agent creation request at top-level (through one of the start commands), or, recursively, when the behavior closure is invoked from a running process.

Making a behavior closure into a process consists of attaching some information to the closure; constituted of a store, a "position", that locates the new process among the system of running processes (giving its concurrence status among other processes), and a "communication environment", that associates with every port the queues of processes waiting for input and for output through this port.

Starting a process consists of making it run in parallel with all already running processes, executing the code in its behavior closure, in the context made of the environment captured by the closure, plus the current communication environment and position information, and, depending on the parameters of the start command, the current store or a distinct copy of it. Stores, positions and communication environments are inherited by processes. This execution may result in a communication, and will eventually produce a new behavior closure, which will be immediately started with the code and environment it has captured, and so on, recursively, until the process terminates, following execution of the `nil` construct, or aborts, as the result of an untrapped exception.

Two user interfaces are provided for LCS: a simulator and an executive, both interactive and including an incremental compiler. The executive interface allows the user to start processes either in background, or in foreground; agents ultimately operate on files, implemented as files of the host operating system, and for which an interface is provided in

distinct, and new, type variables to each label not bound in the prefix. These types are automatically inferred at compile-time.

LCS allows to compute communication ports, in some sense, while preserving the possibility of static type checking. Ports in LCS have two components: a label and a tag. Labels may be hidden or renamed, but may not be computed; on the other side, tags, which can be of any type for which equality is defined, may be computed or passed as messages or parameters, but may not appear in hidings or renamings. Hidings and renamings apply to all ports with the labels involved, collectively. All LCS ports have a tag, the default tag is the value "() : unit", and needs not be mentioned.

LCS allows to create and assign references (pointers) in agents; each running instance of a behavior (i.e. each process) is associated with a store in which side-effects are performed. LCS has both light and strong processes; light composition combinators ($/\backslash$, $\backslash/$) produce processes that share their stores, strong compositions ($//\backslash$, $\backslash\backslash/$) produce processes with private stores. Process stores are inherited at process creations. Two important issues in implementing these aspects are those of atomicity (determines the effects of concurrent store updates) and of reference passing (meaning of). LCS does not protect references from concurrent updates, but forbids reference passing.

Typechecking

Port tags are omitted from the discussion here. Behavior variables stand for infinite type assignments to port labels: $\{p1:'a\}\{p2:'b\}\{p3:'c\}$ etc., in which every label is ascribed a type variable distinct from those assigned to all other labels (and distinct from all variables assigned to labels by other behavior variables). The behavior type in which label p has type t and other labels have the same type as in the behavior type b is written $\{p:t\}b$. In types $b1=\{p:t1\}b$ and $b2=\{p:t2\}b$, every label has the same type in both $b1$ and $b2$, except label p which has type $t1$ in $b1$ and has type $t2$ in $b2$. Unconstrained behavior types sharing no type variables will be denoted by distinct behavior variables.

In expressions where labels are assigned several types (e.g. $\{p:t\}.$), the leftmost assignment supersedes all others; two behavior types are equal if every label is assigned the same type in both behavior types; according to the "leftmost assignment" rule. The concepts of substitution (of a variable by a term), instantiation (of free variables) and of generic instance (instantiation of bound variables) can be extended to terms containing behavior variables; provided behavior variables are substituted by behavior types only.

The LCS typechecker uses unification, as do ML typecheckers, for inferring types for expressions including behaviors. A non trivial extension of the unification algorithm is required for unifying behavior types (as these have the meaning of infinite type assignments and obey a non standard equality). Behavior types unification relies on the following theorem: if it can be assumed that behavior variable \mathbf{X} assigns type variable v to label p , then $\mathbf{X} = \{p:v\}\mathbf{X}$.

In unifying behavior types $\{p:t1\}\mathbf{X}$ and $\{q:t2\}\mathbf{Y}$, for instance, the following substitution would be returned:

$\mathbf{X} \rightarrow \{q:t2'\}\mathbf{X}'$	expansion of \mathbf{X}
$\mathbf{Y} \rightarrow \{p:t1'\}\mathbf{Y}'$	expansion of \mathbf{Y}
$t2' \rightarrow t2$	unification of $t2'$ and $t2$ (for label q)
$t1' \rightarrow t1$	unification of $t1'$ and $t1$ (for label p)
$\mathbf{X}' \rightarrow \mathbf{Y}'$	unification of behavior variables \mathbf{X}' and \mathbf{Y}' (for all labels)

2.0 The Language

Syntax and Constructs

The ML subset of LCS is constituted of a wide subset of Standard ML. The language of agents is constituted of a set of constructs for building behaviors (input, output, etc.), with their own concrete syntax, and a set of standard constructs (abstraction, conditional, etc.), with a syntax identical to the similar constructs of Standard ML.

```
smlexp ::= <sml expressions> | beh agent
agent  ::= id | agent smlexp | fn pat => agent { | pat' => agent' }n
        | agent : ty | let dec in agent end | exp smlexp | behavior
behavior ::= nil | => agent | port ! smlexp => agent | port ? pat => agent
        | agent { / \ } agent' | agent { V | \V } agent' | agent "{ {labels}/labels}"
port ::= label { # smlexp }
pat ::= <sml patterns>
```

Table 1. The (bare) syntax of agents

A new derived binder is introduced: `agent`; `agent` declarations obey the same rules as SML `fun` declarations, except that their bodies are parsed as agents, and that no abstract parameters are required (behaviors may be recursively defined). LCS commands include all those one could type at the top-level of an SML system, plus the derived `agent` declaration, and commands which pertain to the user interface in use: `agent` creation and management at the executive top-level, or simulation commands at the simulator top-level.

LCS supports all CCS constructs for building behaviors. To this set of constructs are added a set of derived constructs, plus tagged ports and strong compositions combinators, soon to be described. One can freely pass agents as arguments, or as messages, to other agents; behaviors enjoy the same rights as other values. `/\` and `\/` denote the CCS parallel and sum compositions, respectively; sums may be unguarded. Renamings and hidings may apply to several labels, simultaneously. Communication ports have the CCS semantics and scope rules; `port` are not denotable, and have global scope, unless restricted. This treatment of communication ports, and, as a consequence, the typing of agents in LCS, are different from those retained in PFL[Hol83][Mit86], another extension of ML with CCS processes, in which ports are particular values passed as parameters to behaviors.

```
@agent mapodds f =
@   let agent maps f = inp? x => out! (f x) => maps f
@   agent gen f x = out! x => gen f (f x)
@   agent pipe a b = (a {tmp/out} /\ b {tmp/inp}) {/tmp}
@   in pipe (gen (fn x => x+2) 0) (maps f)
@   end;
mapodds = - : (int -> 'a) -> {out: 'a}.
```

The set of ML types is enriched with an unbounded set of "behavior types". These may be understood as functions assigning a type with every possible port label. Behavior types have a prefix, in which possibly polymorphic types are assigned to some labels, and a queue, denoted by a sequence of periods, which may be seen as a function assigning

Implementing CCS, the LCS experiment

A SUMMARY OF THE PROJECT

Bernard Berthomieu
LAAS-CNRS*

Keywords: parallel programming languages, language design and implementation, CCS, Standard ML, strong and light processes, polymorphic typechecking techniques, type inference, abstract machines for concurrent programs, virtual machines, garbage collection, orphan processes elimination.

1.0 The LCS Project

The LCS project began in 1985, aimed at studying implementation of modern parallel programming formalisms. This goal was to be achieved through the design and implementation of a language that would have at least CCS [Mil80] capabilities for parallel programming, and be as ergonomic as the most advanced functional languages, such as ML [Gor79]. This short note summarizes this design and implementation experiment.

The language LCS and its implementations evolved, through several versions [Ber85][Ber88], from a simulator running a version of ML extended with a CCS agents layer to the version described here, in which agents, in an extended CCS framework, appear as particular Standard ML values, that may be turned into processes. A non trivial extension of the ML polymorphic typechecking technique, requiring a specific unification algorithm, allows to assign types to agents.

An original abstract machine has been designed and implemented to run LCS programs. Two user interfaces are provided: an executive and a simulator. The executive allows users to start agents in background, while the top-level, seen by the machine as a regular LCS process, collects commands from the user. The LCS virtual machine has, among other peculiarities, that of integrating an automatic allocation and reclamation of memory with both strong and light processes, their stores being managed by copy-on-write.

This summary describes the essential features of the language, and of its implementations; original features of LCS include its typing method for behaviors, some features of the language (tagged ports, strong and light processes, unguarded sums), and many unique features of its implementation. Some familiarity with the essential features of both the language ML and the CCS formalism is assumed.

(*) Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS,
7, avenue du Colonel Roche, 31077 Toulouse Cedex
phone: 61 33 64 08, e-mail: bernard@laas.laas.fr