Implementation of FPL & Concurrency

Advanced Compiler Techniques

2005

Erik Stenman

EPFL

Functional Programming Languages (Repetition)

- Possible properties of a functional languages:
  - No statements.
  - Higher order functions.
  - Pureness.
  - Laziness.
  - Automatic memory management.
- A *declarative language* is a language where the program declares **what** to calculate.
- In an *imperative language* the program states **how** to calculate.

Higher Order Functions (Repetition)

- A function that takes a function as an argument is called a higher order function.
- E.g.

```
def f(x:int, g:int=>int) = x + g(x);
```

Tail calls
(Repetition)

- A function call f(x) within a body of a function g is in a *tail position* if calling f is the last thing g will do before returning.

```
def g(x:int) = f(x + 1);
```

- We can save stack space and execution time by turning the call to f into a jump to f.

Continuations

- We can combine higher order functions with tail calls to get *continuations*.
- Normally each function returns a value:

```
def f(x:int) = foo(x) + 1;
```

- We can instead let each function take a ***continuation*** that tells where the execution is to continue:

```
def f(x:int, c:int=>int) = c(foo(x) + 1);
```

Continuation Passing Style (CPS)

- Continuations are the basis for a compilation technique called *continuation passing style* (*CPS*).
- In CPS all functions are transformed to take one extra argument, the continuation, and the bodies are transformed to call the continuation instead of returning.
- Also, all nested expressions of the function body are transformed into continuations. (Primitive operations such as + also takes a continuation.)

CPS Transformation

```
def f(x:int) = foo(x) + 1;
```

```
def f(x:int, c:int=>int) =
   foo(x,
      (v:int) => +(v, 1 ,c)
      )
```

CPS Transformation

- CPS transformation is used in many compilers for functional languages such as Scheme and ML.
- CPS was studied extensively by e.g. Steele in the Rabbit Scheme compiler, and Appel in the SML/NJ compiler.
- A disadvantage with CPS is that it introduces many closures, and hence the compiler have to optimize away as many of them as possible in order to get good performance.
- An advantage is that, if closures are your only control structure and you have optimized them to the max, then you have **optimized all control structures**.

Implementation of Concurrency

- What is concurrency?
- Some communication methods.
- Erlang – a concurrent language.
- Implementation of Erlang.

Concurrency vs. Parallelism

- Concurrency:
  - If two events are *concurrent* then they **conceptually** take place at the same time. That is, different schedulings of two events are indistinguishable or irrelevant.
  - A **language** can be concurrent.
- Parallelism:
  - If two events occur in parallel then they actually occur at the same time.
  - An **implementation** can be parallel.

Concurrency vs. Parallelism

- A concurrent language can be implemented either in parallel or sequentially.
- Some sequential languages can also be implemented either in parallel or sequentially.
  - Declarative languages are usually easier to make parallel than imperative ones.

Message Passing vs. Shared Memory

- In a concurrent system with *message passing* each message has to be copied from the sender to the receiver. (Like when sending a mail to someone.)
- In a *shared memory* system the participating processes can all updated the shared memory, and the new state is "immediately" visible to all. (Like when two people are writing on and looking at the same blackboard.)

Message Passing vs. Shared Memory

- Shared memory:
  - Pros:
    1. Performance.
  - Cons:
    1. The programmer has to ensure consistency.
    2. Can not (practically) be implemented in a distributed system.
- Message passing:
  - Pros:
    1. Processes are decoupled (errors don't propagate as easily).
    2. The programmer can reason about the process interaction on a higher level.
    3. Can easily be extended to a distributed system.
  - Cons:
    1. (Perceived) loss of performance.

Message Passing vs. Shared Memory

- The distinction between shared memory and message passing is done on the level that the programmer has to deal with.
- On a lower level message passing can be implemented with shared memory (and often is, at least to some extent).
- In a network the shared memory model has to be implemented with some form of message passing.

Synchronous vs. Asynchronous

- In a *synchronous system* both the sender and the receiver have to be in special states (ready to send and ready to receive).
  - If either of the processes reaches this state before the other it will block and wait until both are in the right state.
- In an *asynchronous system* the sender does not have to wait for the receiver to be ready in order to send its message.

Synchronous vs. Asynchronous

- Only one type of primitives is necessary since each can be implemented by the other.
- To implement synchronization in an asynchronous environment you only need a loop and a protocol where an acknowledgement is sent back upon receive.
- To implement asynchronous messages in a synchronous environment you need a relaying process.

Processes vs. Threads

- **In this presentation** *processes* do not refer to OS processes but processes implemented by a programming language.
  - Such processes can be assumed to be lightweight, not to share memory, and execute concurrently.
- A *thread* is slightly more heavyweight, share memory and can execute in parallel on a parallel machine.

Concurrency in Programming Languages

- Concurrency in programming languages can be implemented by utilizing processes or threads from the operating system.
  - Either directly like in C or with a thin abstraction layer like in Java.
  - Further abstractions can be built into libraries.
- Another approach is to build concurrency into the language as such.

Implementation of Concurrency Example: Erlang

- Erlang is a concurrent programming language, i.e., concurrency is built into the language from the beginning.
- Erlang was developed by the Ericsson to be used in large telecom application such as telephone exchanges. (Used in e.g. Ericsson's ATM switch and their GPRS systems.)
- We will present some details of how to implement a concurrent language by studying how Erlang is implemented.

Erlang

- The sequential part of Erlang is a small higher order functional language with no mutable data structures.
- Data in Erlang is represented by a *term*, a term can be a *list* of terms, a *tuple* of terms or ground (*atoms*, *numbers*, *PIDs*, …).
- Erlang uses *pattern matching* to decompose and switch on the structure of Erlang terms.
- Erlang **requires** proper tail-calls.

Erlang

- The concurrent part of Erlang (processes that communicate through message passing) provides the following constructs:
  - Asynchronous send.
    `Receiver ! Message`
  - Blocking, selective receive with timeouts.
    `receive PATTERN -> … ; after T -> … end.`
  - A method to dynamically spawn new processes.
    `spawn(Closure).`
  - For error correction processes can be linked in order to receive signals when a linked process dies:
    `link(Process).`
    or
    `spawn_link(Closure).`

A Simple Generic Server

```
loop(State,Handler) ->
  receive
    {From, Request} ->
      {Res,NewState} = Handler(State,Request),
      From ! {self(), Res},
      loop(NewState,Handler);
    {swap_code,NewHandler} ->
      loop(State, NewHandler);
    quit -> ok
  end.
```

```
  > Server = spawn(fun()->loop(0,
                              fun(S,inc)->{ok,S+1};
                                 (S,get)->{S,S} end)
                 end),
    Server ! {self(),inc}, receive {_,_} -> ok end,
    Server ! {self(),get}, receive {_,Val} -> Val end.
  1
  >
```

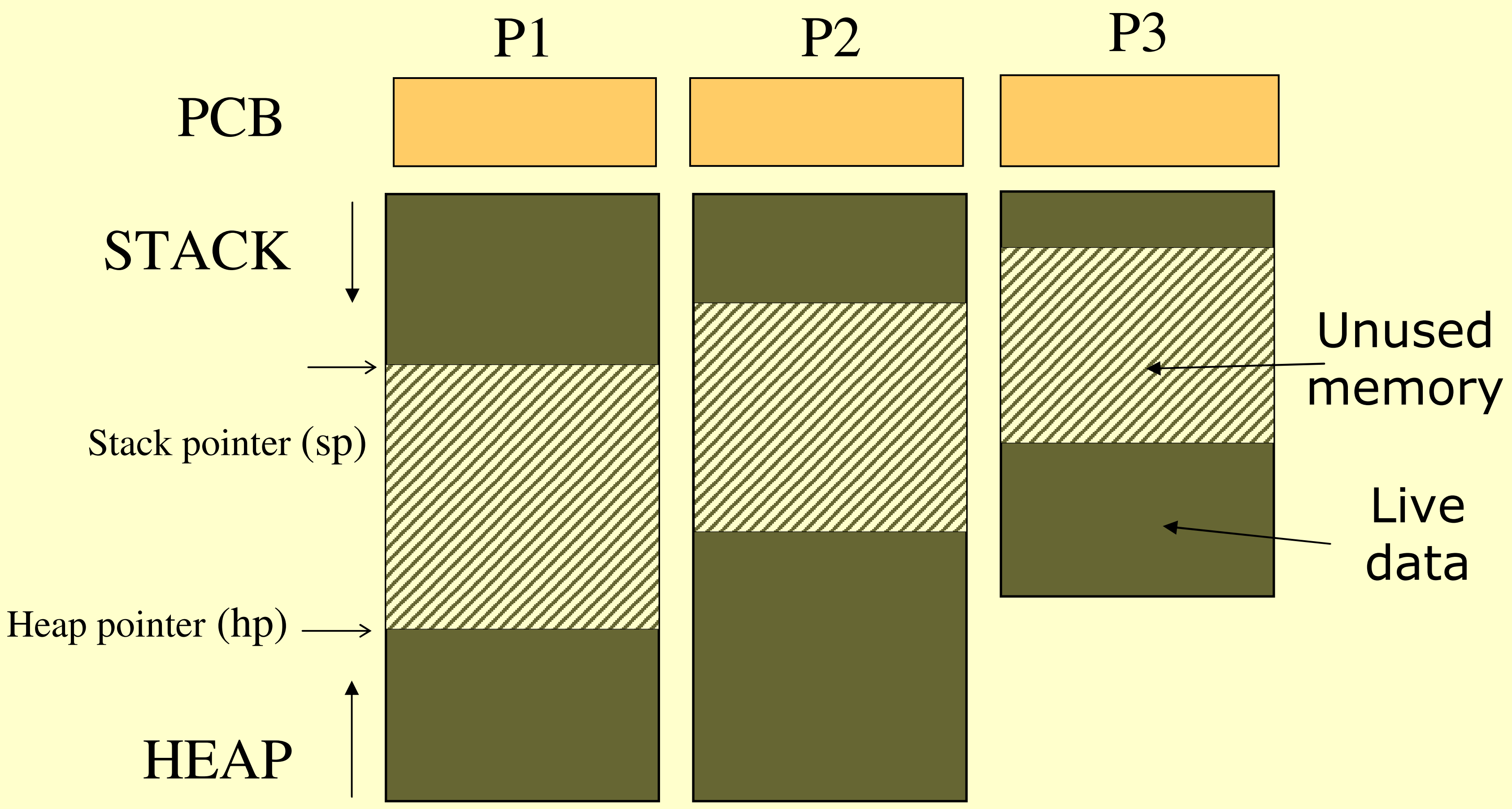
Concurrency in Erlang

- Erlang is *concurrent*.
  - The standard implementation is **not** *parallel*, but multi-tasking.
- Erlang processes are conceptually scheduled with *pre-emptive multitasking* – the programmer does not have to worry about the scheduling.
  - The standard implementation uses *cooperative multitasking* enforced by the compiler.
  - Each function call is counted as a *reduction*, when the number of reductions allocated to a process reaches 0 the process is suspended.
  - Since there are no loop constructs in Erlang other than tail calls, this is sufficient to ensure cooperation.

Implementation of Processes in Erlang

- Each process has its own stack, heap, message queue, and process control block (PCB).
- The PCB is small: ~70 words.
- The mailbox is a linked list of pointers to the heap containing only unprocessed messages.
- The heap and the stack are collocated in one memory area with a default initial size of 233 words. (233=fibonacci(12)).
- The heap and stack grow (and shrink) as needed.

Processes in Erlang



Process Communication in Erlang

- All communication between processes in Erlang is done by *message passing*.
- In the standard implementation this means that all messages are copied between the heap of the sender and the heap of the receiver.
- This copying is done by first calculating the size of the message, then allocating the right amount on the receivers heap, finally the message is copied.
- Since the receiver is guaranteed to be suspended, no locking is needed.

Some "Optimizations"

- Large chunks of immutable data can be stored in *binaries*.
  - Binaries larger than 64 words are not stored on a process heap and not copied when sent as messages.
  - Binaries are managed by reference counting.
- Larger sets of *shared, mutable* data are handled by ETS-tables.
  - ETS stands for *Erlang Term storage*.
  - Conceptually an ETS table could be implemented as a process mapping keys to values.
  - In reality ETS tables are implemented in C.

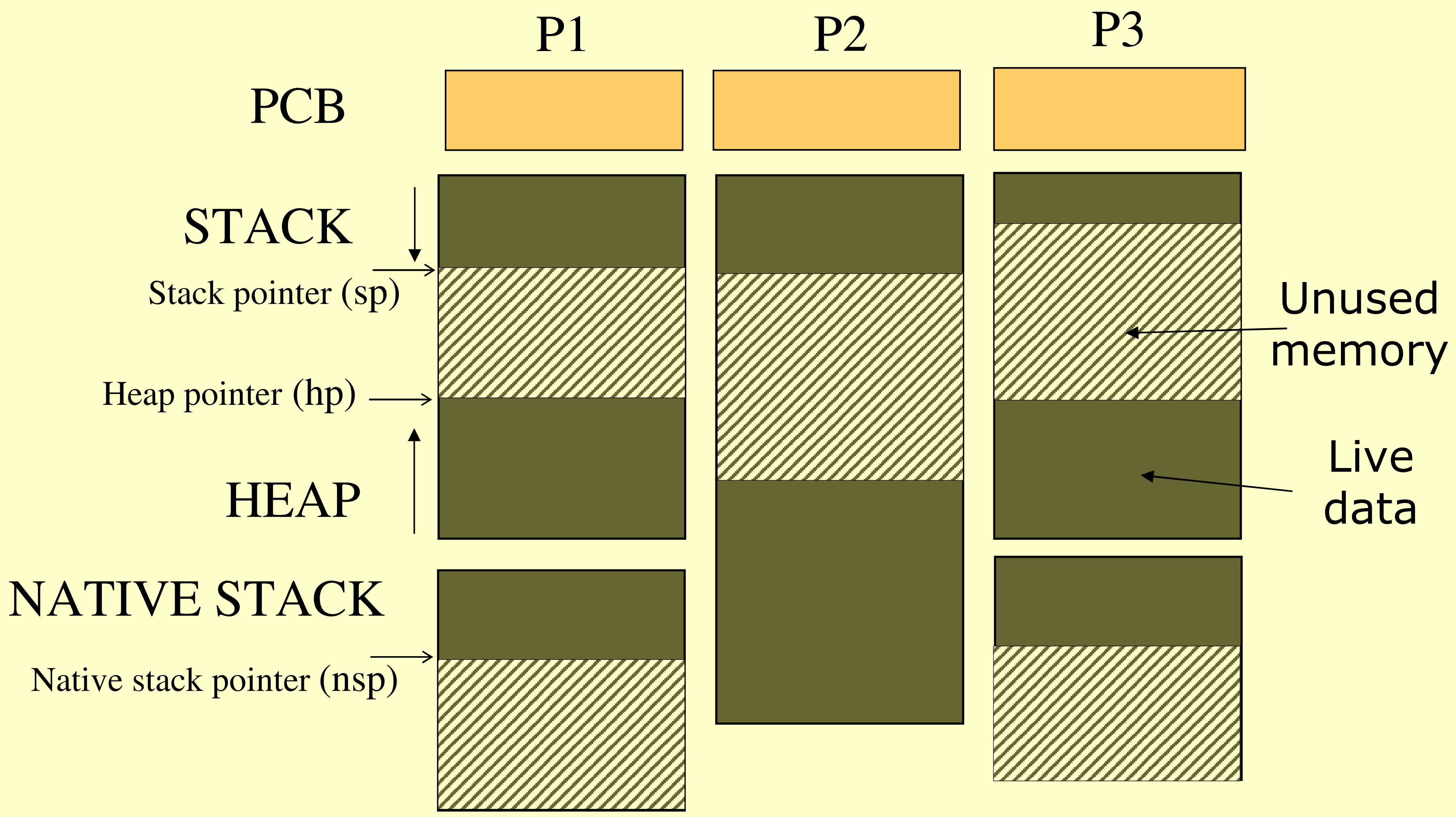
Implementing Erlang in Native Code

- The standard implementation of Erlang uses a virtual machine (VM). We will discuss how to implement VMs in a later lecture.
- It is also possible to compile Erlang to native code, here we will present some implementation details for such an implementation.

Implementing Erlang in Native Code

- In order to enable easy integration with the VM the native implementation uses the same data representation, GC, and runtime system as the VM.
- The only major difference is that each process that calls native code also get a native stack.

Processes in Erlang



Implementation Details

- In order to handle scheduling and stack resizing some bookkeeping code is added to the beginning of each function:

```
reductions = reductions - 1;
if (reductions == 0) suspend(p); // p is the current process pointer
checkstack:
if (nsp - STACKNEED < stackEnd) {
  resizeStack();
   goto checkstack;
}
```

Implementation Details

- The stack need can be calculated at compile time:
  number of spills + max(∀ calls: argsOnStack+callerSaves)+buffer.
- By ensuring that there is a buffer of free words on the stack we do not need the bookkeeping code for leaf-functions that uses less than that many words.

Implementation Details

- The function suspend has to be implemented in machine code in order to get access to the return address.

```
supend:  // p (the current process) is passed as the argument.
    p->pc = <RETADDRESS>  // From the stack on x86 from a register on SPARC
    p->status = READY;
    SAVE(p);    // Save the process sp, switch to C stack.
    add(p,readyQueue);
    p = schedule();
    RESTORE(p);  // Restore the process sp, switch from C stack.
    jmp p->pc;
```

The Scheduler

- Since Erlang does not use OS processes or threads, the Erlang runtime system has to implement its own scheduler. (In, e.g., C)

```
pid schedule() {
  static int majorReductions = MREDS;
  majorReductions--;
  if(majorReductions == 0) { externalPoll();
   majorReductions = MREDS; }
  checkTimeouts();
  pid p = nextReady(readyQueue);
  p->reductions = REDS; p->status = RUNNING;
  return p;
}
```

Send

- A message send from p1 to p2 can be implemented as:

```
send(pid:p1, pid:p2, term:message) {
  int s = size(message);
  if(s > (p2->heapTop - p2->heapPointer)) gc(p2,s);
  term mp = copy(message,p2->heapTop);
  add(mp,p2->messageQueue);
  if(p2->status == SUSPENDED) {
    p2->status = READY;
    add(p2,readyQueue);
  }
}
```

Receive

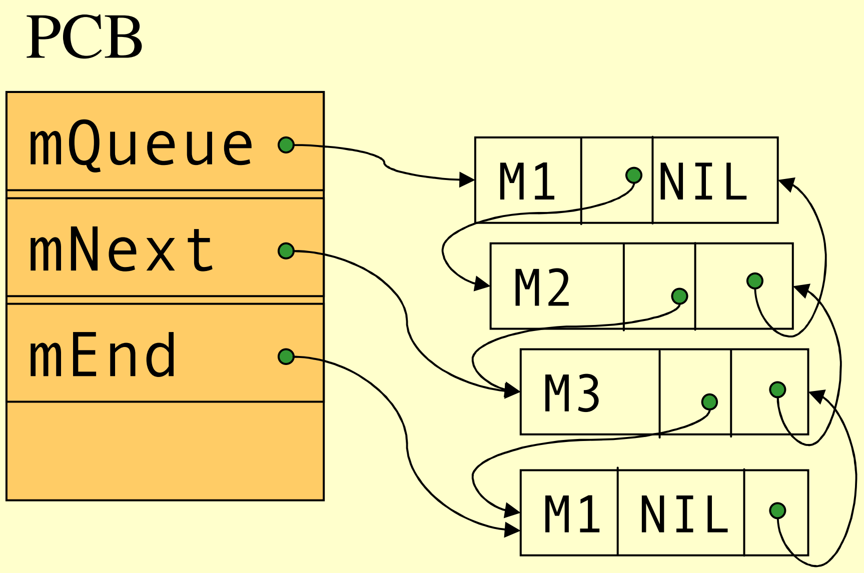
- A message receive is slightly more complicated.

```
messageLoop:
 m = nextMessage(p);
 if (m == NIL)
  sleep(p,timeout,&messageLoop,&handler);
 cont = MATCH(m,PATTERNS);
 if (cont == 0) goto messageLoop;
 unlink(p);
 jmp cont;
handler:
…
```

Receive

```
nextMessage(pid p) {
  term m = p->mNext;
  p->mNext = m->next;
  return m;
}

unlink(pid p) {
  term m = p->mNext;
  if(m->prev != NIL)
    p->prev->next = m->next;
  else
    p->mQueue = m->next;
  if(p->mEnd == m)
    p->mEnd = m->prev;
  p->mNext = p->mQueue;
}
```



Receive

```
sleep(p,timeout,messageLoop,handler) {
  p->pc = messageLoop;
  p->handler = handler;
  add(p,now()+timeout,timeoutQueue)
  p->status = SUSPENDED;
  p = schedule();
  (p->pc)();
}
```

Receive

- The `checkTimeout` function in the scheduler will activate a process when the timeout has elapsed.
- While doing so `p->pc` will be updated with `p->handler` so that the process will start executing in the timeout handler when scheduled.

Spawn

- The spawn primitive creates a new process, i.e. allocates a new PCB, stack, and heap.
- Then the argument to spawn (the closure) is copied to the new heap.
- The new pid is added to the ready queue.
- Then execution continues in the old process with the instructions after spawn.

Summary

- Concurrency is an important concept that can be useful as an abstraction when decomposing a program, just as modules, objects, and functions.
- Concurrency can be implemented by either using primitives provided by the OS or by implementing a scheduler specifically for the language.

Call with Current Continuation `call/cc`

- If we have a language compiled with CPS we can easily implement a very powerful construct called *call/cc* or call *with current continuation.*

```
def call_cc(f,c) = f(c,c)
```

- That is, we call the function f with the current continuation c as an argument, and also as the continuation of f.
- call/cc can be implemented even in non-CPS compilers, but it is trickier because it requires capturing of the stack.
- With call/cc you can "easily" implement backtracking, exceptions, coroutines, and concurrency.