The Pact Smart-Contract Language

Stuart Popejoy <stuart@kadena.io> @SirLensALot

Revision v1.5

June 2017

Introduction

Pact is a new programming language for writing *smart contracts* to be executed by a blockchain, namely Kadena, the breakthrough solution for private blockchains. Pact empowers developers to implement robust, performant transactional logic, executing mission-critical business operations quickly and safely.

Blockchain adoption by industry represents a major transformation in business computing, with organizations eager to reap the benefits of perfect replication and high availability, cryptographically-verified transactions, and lucid, robust business logic. However, implementation has been hindered significantly by the problems with existing smart contract languages. While Bitcoin scripts are nowhere near powerful enough to write full applications, Ethereum's Solidity language and underlying virtual machine are dangerously un-constrained.

Pact is designed from the ground up to support the unique challenges of developing solutions to run on a blockchain. Code is stored in an unmodified, human-readable form on the ledger. Pact's declarative, immutable and Turing-incomplete design fights exploits and bugs. Excellent development tooling speeds development and assists troubleshooting. With Pact, blockchain can at last realize its full transformational potential.

This paper discusses the design background of Pact, including a discussion of *transactional* versus *analytic workloads* on blockchain. Pact's safe-computing features are discussed, as well as its notion of *encapsulating* database tables in Pact code modules. It introduces *keysets*, Pact's approach to signature verification and authorization. Finally, a small code example is provided detailing a simple account-balance use-case.

Key Features

- Turing-incomplete safety-oriented design
- Human-readable, on-ledger code
- Atomic execution (transactions)
- Module definition and import
- Unique "key-row" + columnar database metaphor
- Expressive syntax and function definition
- Type inference
- "Module-guarded" tables
- Single-sig and multi-sig public-key authorization
- Multi-step "pacts" for confidential execution
- Key rotation support
- Designed for direct integration with industrial databases

Pact Smart Contracts in brief

A smart contract in Pact is comprised of three core elements: tables, keysets, and a code module.

Contract Tables

Data in Pact is stored in tables, which have a "key-row" structure, support schemas, and support a versioned, columnar history.

Contract Keysets

Pact keysets group a set of public keys with a *predicate function* to comprise a "key validation rule" for authorization. Keysets can be defined globally or at a per-row level. All modules are governed by an "administrative" keyset.

Contract Module

A *module* is where all code is declared implementing a smart contract, comprised of function definitions, *pact* definitions (multi-step functions for confidential execution), tables and schemas. The module defines the API of the smart contract, and "guards" all access to contract tables.

Executing Contract Transactions

Contract operations are executed by sending in one or more module function calls. A given blockchain message comprises a single *transaction* in the database sense of *atomic execution*: if any failure occurs, the entire operation is rolled back to the state before the function was executed.

Querying Contract Data

Queries in Pact are focused on data-export instead of heavy analytical processing. For best performance, queries should be executed in the blockchain's "local mode"; most historical queries take a *transaction ID* which guarantees the validity of the data up until that transaction ID in the blockchain.

Design Background

Pact's design is informed by existing approaches to smart contracts, namely Bitcoin and Ethereum, as well as stored procedure languages, SQL, and LISP.

Bitcoin Contracts

Smart-contracts in blockchain begin with Bitcoin, which validates each transaction by executing a "script" written in a minimal, non-Turing complete bytecode language.

Bitcoin scripts are not powerful enough to function as full "smart contracts" in the Pact or Ethereum sense, as they can only pass-or-fail a transaction: the actual transfer happens in hardcoded Bitcoin software. With smart contracts the intent is to both verify and *enact* the full logic of a given transaction.

Bitcoin script nonetheless inspires Pact's design. Bitcoin's signature verification, as performed by the Bitcoin `CHECKSIG` and `OP_CHECKMULTISIG` opcodes, informs Pact's keyset-verification scheme. Pact is also informed by Bitcoin's principled rejection of Turing-completeness, with its associated risks of exploits and bugs.

Ethereum

Like Bitcoin, Ethereum offers a bytecode language for executing transactions, but greatly expands its power: it specifies a full virtual machine (EVM) to execute its rich, Turing-complete instruction set. Within the EVM, contracts can write to long-term storage, access short-term memory locations, call other contracts (with recursion), and loop indefinitely, limited only by a "gas" computational cost model.

Turing-complete considered harmful

EVM's unrestricted computational power poses very real risks, amply illustrated by the recent "DAO hack" of $50m of Ether, exploiting a loophole in the bewilderingly complex logic of the DAO smart contract. Simply put, Turing-complete computation within a money-handling transactional environment is inappropriate and dangerous.

Pact is immutable, Turing-incomplete and favors a declarative approach over complex control-flow. This makes bugs harder to write and easier to spot. Pact smart contracts are designed to enforce business rules guarding the update of a system-of-record: complex, speculative application logic simply does not belong in this critical layer.

Stored Procedures and SQL

Industrial RDBMS systems have long supported *stored procedures* as a way to house business logic inside a database. Smart contracts are often compared to stored procedures; their similarities and differences merit examination.

Usage of Stored Procedures vs "just SQL"

From a functional perspective, stored procedures are often unnecessary, as SQL has the power to perform most database operations. Thus, the use of stored

procedures signals a more rigorous approach to database API design: as stored procedures are persisted in the database, they standardize business logic and encourage code reuse. Moreover, stored procedures can "guard" the database against users issuing arbitrary SQL statements that might harm system performance or violate data integrity.

Smart contract languages like Pact and Ethereum's Solidity resemble stored procedures in this respect. Code is deployed into the blockchain, exposing functions that house the detailed transaction logic. Transactions are executed by sending in brief snippets of code that call exposed functions with appropriate argument values.

Ethereum and Pact smart contracts both "guard" data in their own way. Ethereum contracts have a dedicated "namespace" in permanent storage that cannot be accessed outside the contract. In Pact, one or more tables can be "guarded" by a module, preventing direct access to the table, and allowing the module to define a coherent and safe data-access API.

Code as Data

With stored procedures, code is deployed administratively, with no need to support versioned histories of that code on the database. With blockchain, we see a very different model emerge. For Bitcoin, which persists the execution bytecode with each transaction, storing code is critical to the notion of a transparent execution chain: if you cannot verify the script, you cannot be sure the transaction was identically executed everywhere.

Thus the model of "code-as-data" emerges in blockchain, where the code to be executed is as important to the transaction as the business data itself. Ethereum similarly stores bytecode on the ledger for every contract and transaction, allowing for quick execution: bytecode requires no translation or compilation to execute. However, there is no way to reliably verify what code is indeed running on the chain: Solidity compiles to huge volumes of impenetrable bytecode.

Pact code is stored directly on the ledger, human-readable and easy to verify. The usage of LISP syntax ensures the code parses and executes as fast as possible. LISP's *s-expression* syntax parses quickly and directly models the syntax tree (AST) to be executed by the machine. Pact's functional, non-Turing-complete approach encourages *shorter programs*. The original code executes

directly on the ledger, and can be verified by human eyes as part of an immutable transaction history.

As a result, Pact is an extremely "lucid" computing environment, which is further amplified by its tooling. Compiler and runtime errors offer detailed information, with stack traces and function documentation. Pact ships with a feature-rich REPL, allowing for rapid, iterative development, with incremental transaction execution, and environment and database inspection. Developing Pact code is fun and productive.

Atomic Execution

RDBMS systems offer *atomic execution:* data-updating code runs inside of a "transaction" which *commits* any changes only upon success of the entire code path. Any errors encountered automatically *roll back* any changes to the database before aborting execution.

Support for atomic execution is a hard requirement for transactional database environments. Expecting programmers to manually "clean up" after any error scenario is not just foolhardy, but impossible in some scenarios. In Pact, smart-contracts are executed atomically: updates are only persisted upon successful completion of the code.

Transactional vs. Analytic Workloads

Most use-cases for blockchain are *transactional.* This is clearly true for Bitcoin and other ledger-oriented solutions, but also for new use-cases for blockchains like medical record transfer and storage. The goal is to implement a *system of record* that can immutably and permanently enact, capture and replicate the results of some business event.

We characterize this as a *transactional workload,* requiring robustness and high performance over whatever volumes the business case requires. In database terminology this would be identified as an *OLTP system* (online transaction processing) as opposed to an *OLAP system* (online analytic processing). OLAP systems, or *data warehouses*, have radically different operational requirements: they must support efficient querying on non-key attributes, and speedily execute operations like aggregates, filters and joins.

Best practices in databases dictate the clear separation of OLTP and OLAP designs. OLAP seeks thorough normalization, strong typing (schemas), foreign-key constraints, and extensive indexing. In an OLTP context, these features are unnecessary and harm performance, especially indexes. OLTP best practices avoid indexing and joins, prefer denormalized ("fat") tables, and discourage foreign-key constraints and complex schemas.

OLTP and Distributed Ledger

Most OLTP systems regularly export data to an OLAP warehouse. The OLTP system itself has little need for historical data, and large datasets can negatively impact performance. Thus, after export, the data is usually deleted: regular "cleaning" of OLTP tables is a best practice.

Blockchains on the other hand famously keep all data, forever. This can be seen as a drawback for industrial applications, based on the Bitcoin experience of fresh client installs chewing through 75GB+ (and growing) of transaction data before coming on-line. At Kadena we believe this problem is easily mitigated through standard database techniques: Kadena supports *checkpointing* of the smart-contract database, allowing for efficient replay and startup. Properly managed, the permanence of a blockchain is a benefit, as it eliminates the worries of replication, disaster recovery, and high-availability.

What blockchain does *not* provide, however, is a performant OLAP solution. We believe this *analytic workload* is not suitable for a blockchain, and encourage regular exporting of smart-contract data to "downstream" systems to support analytic use-cases. Pact is designed to allow direct integration with an industrial RDBMS if desired for efficient publication of historical data.

Beyond OLTP

Blockchain encompasses more than just event-capture. The ability to cryptographically "prove" correctness encourages adding validation of key invariants into the transactional workload. As a result, data access is required, moving away from a "pure" OLTP write- or append-only approach.

Bitcoin is a good example of the limitations of a pure-transactional data model. While UTXO-based transactions are terrifically robust, it puts a significant

burden on an upstream system to specify what outputs to spend in a particular transaction, requiring an up-to-date "wallet" database.

A smart-contract language should provide "just enough" query functionality to provide a full solution. To this end, Pact provides a hybrid data model combining columnar, relational and key-value approaches. Tables have a "key-row" interface, with rows having multiple columns values. Smart contracts read and write the "latest value" of a table entry, while the database maintains a full versioned, columnar history.

To work with data at some key, contracts use the `with-read` binding construct to populate variables with specified column values. Contracts write data using JSON-like object syntax to update multiple column values at once. Computation on historical data is more constrained. While Pact offers some aggregation capability via functions like `filter` and `fold`, it does not offer joins, and in general is intended to facilitate data export, not analysis.

Safe Computing in Pact

With LISP-like syntax, user functions and modules, and imperative style, Pact resembles a general-purpose, Turing-complete language. However, to support the design goal of "just enough" power for transactional blockchain solutions, Pact has deliberate constraints on its computational ability.

Immutability

Variables in Pact are "bound," not "assigned", using the standard LISP `let` as well as the special `with-read` and `bind` forms. Variables cannot be changed or reassigned, which boosts the safety and comprehensibility of Pact code.

No Unbounded Looping or Recursion (Turing-incomplete)

Recursion in Pact is detected and causes an immediate failure at module load. Looping is only supported using `map` and `fold` on finite list structures.

A benefit of this restriction is that Pact does not need to employ any kind of cost model like Ethereum's "gas" to limit computation. The lack of recursion

also means Pact can aggressively "inline" function calls at module load time, boosting performance.

Conditionals

Pact has the standard `if` statement for branching execution. However, authors are encouraged wherever possible to instead use `enforce` to verify invariants, aborting the transaction if not met.

Type Safety

Pact added *type inference* in version 2.0, making it possible for module code to be strongly-typed without having to annotate every declaration with type information; tables gained the ability to have a *schema* defined to extend type safety to the database. Inference has the added benefit of limiting run-time type enforcement: developers can use the `typecheck` to add "just enough types" to eliminate warnings and only enforce types at runtime where needed. Often, and similarly to SQL, table schemas provide enough information alone to typecheck an entire module!

No Nulls

Pact does not support null values in code or in the database. This is a significant departure from SQL and RDBMS tradition (ironically, as `NULL` famously violates the relational calculus). Indeed, `NULL`s violate *totality* by allowing authors to dodge handing scenarios resulting from missing data, unhandled edge-cases, or excessive denormalization.

Pact's `with-read` function binds variables to database columns by name. If a column is not present in the data, the transaction fails. This forces module authors to be *total* in how they model entity attributes: attributes must have a value if Pact is to compute on them. Authors can employ the `with-default-read` function to provide default values if a row is not present.

No lambdas, macros, or `eval`

Departing from LISP, Pact does not support lambdas (anonymous functions): authors can only declare module-level functions only. This is primarily for performance but also avoids creating closures which can significantly complicate variable scoping, leading to bugs, or implement recursion (using a Y-combinator). Likewise, Pact modules cannot evaluate arbitrary strings, or run macros, for safety and simplicity.

Modules and Tables

Smart contract logic is defined in a *module* built from function declarations and table specifications, which are then invoked in transactional code and other modules. A key feature of modules in Pact is to *encapsulate data access* by "guarding" tables.

Tables are defined within module declarations using the `deftable` keyword. Access to this table is governed by the module: direct access to the table is limited to administrators, guarded by the module's administrative keyset. However, code declared *within* the module has unlimited access.

The result is to empower module authors with complete control over the authorization, representation and manipulation of data within transactions. "Normal" line-of-business access to the table can only occur by calling module functions. Authors can leverage Pact keysets to provide whatever user-level authorization schemes necessary. Meanwhile, administrative access allows for data migration and emergency procedures.

Keysets and Transaction Signing

A major feature of Pact is *keysets*, Pact's public-key signature authorization support. Pact is designed for use with a public-key signing scheme like Bitcoin, where incoming transactions are signed with a provided public key or keys.

Before executing any code, the Pact runtime first verifies all signatures, with verification failure aborting the transaction. On success, the execution

environment is populated with the verified keys. Pact code can then match these verified keys against a predefined rule called a *keyset*, which combines a defined set of public keys with a *keyset predicate function*.

The keyset predicate function specifies logic determining how transaction keys may match against the keyset. The trivial use-case of a single key obviously needs only one match, while diverse multi-sig schemes are easily modeled: match all, match at least one, match a majority, and so on.

Key Rotation

Keysets inherently support *key rotation.* Initial definition of a keyset requires no special permissions. Any *re-definition* of a keyset, however, requires the transaction to verify against the existing, old keyset and predicate. In this way keysets can be securely modified to remove or add keys, or to change the predicate function.

Row-level Keysets

Keysets can be stored in the database as normal data, enabling *row-level authorization*: contract code can access a row to read its keyset and enforce it before proceeding. This enables authorization use-cases like Bitcoin's Pay-To-PubkeyHash, where account access is individually authorized by distinct keys. Row level functionality is thus "multi-sig ready" as well, and supports rotation by the mechanism described above.

Confidential Computing with "Pacts"

In privacy-preserving blockchain where participants can only decrypt and run a subset of smart contract executions, we see the Pact database becoming *disjoint* between different "entities" (groupings of participants, i.e. a company, or group), as it is necessarily composed of different datasets each.

This presents problems for traditional Blockchain contracts, which expect to write "all sides" of a transaction. For instance, a payment transfer use case can no longer execute a single code block to move funds from one entity's account to the next.

To resolve this problem, Pact looks to *coroutines* in computer science: functions that can *yield* and *resume* at key points in a single function execution. In Pact, these functions are called *pacts*, which define *steps* to be executed by different entities as sequential transactions on the blockchain.

A payment transfer expressed as a multi-step pact would first debit the payor's account in the payor entity's database, before yielding execution. Successful completion results in a signed "resume message" being sent back into the blockchain. This is detected by the payee's entity node, which would then credit the payee's account in a second transaction. If this step aborts, the debit step can be defined with rollback logic to re-credit the payor account once it sees this second transaction has failed.

Example: an `accounts` module

We present example Pact code, implementing a simple "account balance" smart contract, with functions to create accounts and transfer funds.

```
(define-keyset 'accounts-admin
  (read-keyset "accounts-admin-keyset"))

(module accounts 'accounts-admin
 "Simple account functionality."

 (defschema account
  "Schema for accounts table."
  balance:decimal
  amount:decimal
  keyset:keyset
  note)

 (deftable accounts:{account})

 (defun create-account (address keyset)
  (insert accounts address
         { "balance": 0.0, "amount": 0.0, "keyset": keyset,
           "note": "Created account" }))

 (defun transfer (src dest amount)
   "transfer AMOUNT from SRC to DEST"
  (with-read accounts src
            { "balance":= src-balance
            , "keyset" := src-ks }
   (enforce-keyset src-ks)
   (check-balance src-balance amount)
   (with-read accounts dest { "balance":= dest-balance }
```

```
    (write accounts src
           { "balance": (- src-balance amount)
           , "amount": (- amount)
           , "note": { "transfer-to": dest } })
    (write 'accounts dest
           { "balance": (+ dest-balance amount)
           , "amount": amount
           , "note": { "transfer-from": src } })))))

 (defun check-balance (balance amount)
   (enforce (<= amount balance) "Insufficient funds"))
)

(create-table accounts)
```

Installing the module

This code is deployed as a transaction message, containing the code and JSON data with the admin keyset definition:

```
{ "code": ... above code encoded as JSON string ...,
  "data": {
    "accounts-admin-keyset": {
      "keys": ["a987def98f7d543fad97...","f470bc3cd6ff23d77e7..."],
      "pred": "keys-any"
    }
  }
  ...digest and signer pubkey...
}
```

The *accounts-admin-keyset* allows access using either of the two keys by specifying `keys-any` as its predicate.

Keyset Definition

The `define-keyset` call creates the *accounts-admin* keyset, reading the keyset definition from the JSON transaction payload with `read-keyset`. Invalid JSON or a missing key results in transaction failure.

Module Definition

The `module` keyword defines a module `accounts`, designating *accounts-admin* as its administrative keyset. The module defines a schema, a table, and three functions.

`defschema` creates a schema, here called "account" a user-defined datatype used for tables as well as the objects returned by the table. The subsequent `deftable` form defines the "accounts" table and types it with the "account" schema.

`create-account` inserts a new value into the *accounts* table keyed with the `address` value, inserting the provided keyset, an initial amount and balance, and a note. `insert` fails the transaction if an entry already exists for the key.

`check-balance` calls `enforce`, testing available balance. Pact has the usual comparative and arithmetic operations, with LISP-style prefix syntax.

`transfer` is the main functionality of the module.

1. It calls `with-read` to bind `src-balance` and `src-ks` to the *balance* and *keyset* attributes of the `src` account entry, which fails if the entry is not found, or if the columns are not populated.
2. It enforces the row keyset `src-ks` using `enforce-keyset`, and calls `check-balance` to enforce that the `src` account has `amount`.
3. It reads the `dest` account entry, enforcing its existence. It has now enforced all invariants and is ready to perform the update.
4. The `write` to the `src` account debits the balance, records the amount, and notes the destination account.
5. The `write` to the `dest` account credits the balance, records the amount, and notes the source account.

Table Creation

Finally, the *accounts* table is created and associated with the `accounts` module. Direct access to the table will thus be guarded by the *accounts-admin* keyset, while functions like `transfer` are able to enforce bespoke authorization schemes and access the table as necessary.

Invoking the `accounts` module

Transactional code using the accounts module would come in a signed message with Pact code and JSON data. For instance, to create a new account, a message might look like the following:

```
{ "code":
    "(accounts.create-account \"ABC\" (read-keyset \"keyset\"))"
  "data": {
    "keyset": {
      "keys": ["b67a66e17ddd2c30acd..."],
      "pred": "keys-all"
    }
  }
  ...digest and signer pubkey...
}
```

The code references the `create-account` function from the accounts module using qualified syntax. Transactional code can utilize `use` to import a module's functions but this is less performant. The account *ABC* is created, reading the keyset from the JSON data. If a subsequent call to `transfer` debiting *ABC* does not have this key, the transaction will fail.

Conclusion

Smart-contracts present a new modality of transaction processing with unique requirements and risks, necessitating careful reasoning about what kinds of features should be present and what should be avoided. Nonetheless, history has much to offer, with decades of bulletproof database-backed transactional systems, the original code-as-data LISP environments of the 80s, and Bitcoin itself. We believe Pact represents a powerful synthesis drawn from this rich background which will unleash the private-blockchain revolution.

Revision History

V1.5: Updated for Pact 2.x (types, schemas, deftable/defschema, updated syntax).