Pact Language Reference Documentation

Release 4.2.1

Stuart Popejoy

May 04, 2022

Contents

Contents:

CHAPTER 1

Pact Smart Contract Language Reference

This document is a reference for the Pact smart-contract language, designed for correct, transactional execution on a high-performance blockchain. For more background, please see the white paper or the pact home page.

CHAPTER 2

Rest API

See (https://api.chainweb.com/openapi/pact.html) for latest OpenAPI docs.

2.1 Pact built-in server

Pact ships with a built-in HTTP server and SQLite backend. To start up the server issue `pact -s config.yaml`, with a suitable config.

2.2 `pact-lang-api` JS Library

The `pact-lang-api` JS library is available via npm for web development.

2.3 API request formatter

The `pact` tool accepts the `-a` option to format API request JSON, using a YAML file describing the request. The output can then be used with a POST tool like Postman or even piping into `curl`.

For instance, a yaml file called "apireq.yaml" with the following contents:

```
code: "(+ 1 2)"
data:
  name: Stuart
  language: Pact
keyPairs:
  - public: ba54b224d1924dd98403f5c751abdd10de6cd81b0121800bf7bdbdcfaec7388d
    secret: 8693e641ae2bbe9ea802c736f42027b03f86afe63cae315e7169c9c496c17332
```

can be fed into `pact` to obtain a valid API request:

```
$ pact -a tests/apireq.yaml -l
{"hash":
↪"444669038ea7811b90934f3d65574ef35c82d5c79cedd26d0931fddf837cccd2c9cf19392bf62c485f33535983f5e04c3e
↪","sigs":[{"sig":
↪"9097304baed4c419002c6b9690972e1303ac86d14dc59919bf36c785d008f4ad7efa3352ac2b8a47d0b688fe2909dbf392
↪","scheme":"ED25519","pubKey":
↪"ba54b224d1924dd98403f5c751abdd10de6cd81b0121800bf7bdbdcfaec7388d","addr":
↪"ba54b224d1924dd98403f5c751abdd10de6cd81b0121800bf7bdbdcfaec7388d"}],"cmd":"{\
↪"address\":null,\"payload\":{\"exec\":{\"data\":{\"name\":\"Stuart\",\"language\":\
↪"Pact\"},\"code\":\"(+ 1 2)\"}},\"nonce\":\"\\\"2017-09-27 19:42:06.696533 UTC\\\"\
↪"}"}
```

Here's an example of piping into curl, hitting a pact server running on port 8080:

```
$ pact -a tests/apireq.yaml -l | curl -d @- http://localhost:8080/api/v1/local
{"status":"success","response":{"status":"success","data":3}}
```

2.4 Request YAML file format

Request yaml files takes two forms. An *execution* Request yaml file describes the exec payload. Meanwhile, a *continuation* Request yaml file describes the cont payload.

2.4.1 YAML exec command request

The execution request yaml for a public blockchain takes the following keys:

```
code: Transaction code
codeFile: Transaction code file
data: JSON transaction data
dataFile: JSON transaction data file
keyPairs: list of key pairs for signing (use pact -g to generate): [
  public: base 16 public key
  secret: base 16 secret key
  caps: [
    optional managed capabilities
    ]
  ]
nonce: optional request nonce, will use current time if not provided
networkId: string identifier for a blockchain network
publicMeta:
  chainId: string chain id of the chain of execution
  sender: string denoting the sender of the transaction
  gasLimit: integer gas limit
  gasPrice: decimal gas price
  ttl: integer time-to-live value
  creationTime: optional integer tx execution time after offset
type: exec
```

2.4.2 YAML Continuation command request

The continuation request yaml for a public blockchain takes the following keys:

```
pactTxHash: integer transaction id of pact
step: integer next step of a pact
rollback: boolean for rollingback a pact
proof: string spv proof of continuation (optional, cross-chain only)
data: JSON transaction data
dataFile: JSON transaction data file
keyPairs: list of key pairs for signing (use pact -g to generate): [
  public: string base 16 public key
  secret: string base 16 secret key
  caps: [
    optional managed capabilities
    ]
  ]
networkId: string identifier for a blockchain network
publicMeta:
  chainId: string chain id of the chain of execution
  sender: string denoting the sender of the transaction
  gasLimit: integer gas limit
  gasPrice: decimal gas price
  ttl: integer time-to-live value
  creationTime: optional integer tx execution time after offset
nonce: optional request nonce, will use current time if not provided
type: cont
```

Note that the optional "proof" field only makes sense when using cross-chain continuations.

2.5 Signing Transactions

As of Pact 3.5.0, the `pact` command line tool now has several commands to facilitate signing transactions. Here's a full script showing how these commands can be used to prepare an unsigned version of the transaction and add signatures to it. This transcript assumes that the details of the transaction has been specified in a file called `tx.yaml`.

```
# At some earlier time generate and save some public/private key pairs.
pact -g > alice-key.yaml
pact -g > bob-key.yaml

# Convert a transaction into an unsigned prepared form that is signatures can be␣
↪added to
pact -u tx.yaml > tx-unsigned.yaml

# Sign the prepared transaction with one or more keys
cat tx-unsigned.yaml | pact add-sig alice-key.yaml > tx-signed-alice.yaml
cat tx-unsigned.yaml | pact add-sig bob-key.yaml > tx-signed-bob.yaml

# Combine the signatures into a fully signed transaction ready to send to the␣
↪blockchain
pact combine-sigs tx-signed-alice.yaml tx-signed-bob.yaml > tx-final.json
```

The `add-sig` command takes the output of `pact -u` on standard input and one or more key files as command line arguments. It adds the appropriate signatures to to the transaction and prints the result to stdout.

The `combine-sigs` command takes multiple unsigned (from `pact -u`) and signed (from `pact add-sig`) transaction files as command line arguments and outputs the command and all the signatures on stdout.

Both `add-sig` and `combine-sigs` will output YAML if the output transaction hasn't accumulated enough signatures to be valid. If all the necessary signatures are present, then they will output JSON in final form that is ready to

be sent to the blockchain on the `/send endpoint <#send>‘__. If you would like to do a test run of the transaction, you can use the -l flag to generate output suitable for use with the `/local endpoint <#local>‘__.

The above example adds signatures in parallel, but the add-sig command can also be used to add signatures sequentially in separate steps or all at once in a single step as shown in the following two examples:

```
cat tx-unsigned.yaml | pact add-sig alice-key.yaml | pact add-sig bob-key.yaml
cat tx-unsigned.yaml | pact add-sig alice-key.yaml add-sig bob-key.yaml
```

2.5.1 Offline Signing with a Cold Wallet

Some cold wallet signing procedures use QR codes to get transaction data on and off the cold wallet machine. Since QR codes can transmit a fairly limited amount of information these signing commands are also designed to work with a more compact data format that doesn't require the full command to generate signatures. Here's an example of what tx-unsigned.yaml might look like in the above example:

```
hash: KY6RFunty4WazQiCsKsYD-ovu-_XQByfY6scTxi9gQQ
sigs:
  368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca: null
  6be2f485a7af75fedb4b7f153a903f7e6000ca4aa501179c91a2450b777bd2a7: null
cmd: '{"networkId":"mainnet01","payload":{"exec":{"data":{"ks":{"pred":"keys-all",
↪"keys":["368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca"]}},"code
↪":"(coin.transfer-create \"alice\" \"bob\" (read-keyset \"ks\") 100.1)\n(coin.
↪transfer \"bob\" \"alice\" 0.1)"}},"signers":[{"pubKey":
↪"6be2f485a7af75fedb4b7f153a903f7e6000ca4aa501179c91a2450b777bd2a7","clist":[{"args
↪":["alice","bob",100.1],"name":"coin.TRANSFER"},{"args":[],"name":"coin.GAS"}]},{
↪"pubKey":"368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca","clist
↪":[{"args":["bob","alice",0.1],"name":"coin.TRANSFER"}]}],"meta":{"creationTime
↪":1580316382,"ttl":7200,"gasLimit":1200,"chainId":"0","gasPrice":1.0e-5,"sender":
↪"alice"},"nonce":"2020-01-29 16:46:22.916695 UTC"}'
```

To get a condensed version for signing on a cold wallet all you have to do is drop the cmd field. This can be done manually or scripted with cat tx-unsigned.yaml | grep -v "^cmd:". The result would look like this:

```
hash: KY6RFunty4WazQiCsKsYD-ovu-_XQByfY6scTxi9gQQ
sigs:
  368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca: null
  6be2f485a7af75fedb4b7f153a903f7e6000ca4aa501179c91a2450b777bd2a7: null
```

Keep in mind that when you sign these condensed versions, you won't be able to submit the output directly to the blockchain. You'll have to use combine-sigs to combine those signatures with the original tx-unsigned.yaml file which has the full command.

2.5.2 Detached Signature Transaction Format

The YAML input expected by pact -u is similar to the *Public Blockchain YAML format* described above with one major difference. Instead of the keyPairs field which requires both the public and secret keys, pact -u expects a signers field that only needs a public key. This allows signatures to be added on incrementally as described above without needing private keys to all be present when the transaction is constructed.

Here is an example of how the above tx.yaml file might look:

```
code: |-
  (coin.transfer-create "alice" "bob" (read-keyset "ks") 100.1)
  (coin.transfer "bob" "alice" 0.1)
```

(continues on next page)

(continued from previous page)

```
data:
  ks:
    keys: [368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca]
    pred: "keys-all"
publicMeta:
  chainId: "0"
  sender: alice
  gasLimit: 1200
  gasPrice: 0.0000000001
  ttl: 7200
networkId: "mainnet01"
signers:
  - public: 6be2f485a7af75fedb4b7f153a903f7e6000ca4aa501179c91a2450b777bd2a7
    caps:
      - name: "coin.TRANSFER"
        args: ["alice", "bob", 100.1]
      - name: "coin.GAS"
        args: []
  - public: 368820f80c324bbc7c2b0610688a7da43e39f91d118732671cd9c7500ff43cca
    caps:
      - name: "coin.TRANSFER"
        args: ["bob", "alice", 0.1]
type: exec
```

CHAPTER 3

Concepts

3.1 Execution Modes

Pact is designed to be used in distinct *execution modes* to address the performance requirements of rapid linear execution on a blockchain. These are:

1. Contract definition.
2. Transaction execution.
3. Queries and local execution.

3.1.1 Contract Definition

In this mode, a large amount of code is sent into the blockchain to establish the smart contract, as comprised of modules (code), tables (data), and keysets (authorization). This can also include "transactional" (database-modifying) code, for instance to initialize data.

For a given smart contract, these should all be sent as a single message into the blockchain, so that any error will rollback the entire smart contract as a unit.

Keyset definition

Keysets are customarily defined first, as they are used to specify admin authorization schemes for modules and tables. Definition creates the keysets in the runtime environment and stores their definition in the global keyset database.

Namespace declaration

Namespace declarations provide a unique prefix for modules and interfaces defined within the namespace scope. Namespaces are handled differently in public and private blockchain contexts: in private they are freely definable, and the *root namespace* (ie, not using a namespace at all) is available for user code. In public blockchains, users are not

allowed to use the root namespace (which is reserved for built-in contracts like the coin contract) and must define code within a namespace, which may or may not be definable (ie, users might be restricted to "user" namespaces).

Namespaces are defined using *define-namespace*. Namespaces are "entered" by issuing the *namespace* command.

Module declaration

Modules contain the API and data definitions for smart contracts. They are comprised of:

- *functions*
- *schema* definitions
- *table* definitions
- *pact* special functions
- *constant* values
- models
- *capabilities*
- *imports*
- *implements*

When a module is declared, all references to native functions, interfaces, or definitions from other modules are resolved. Resolution failure results in transaction rollback.

Modules can be re-defined as controlled by their governance capabilities. Often, such a function is simply a reference to an administrative keyset. Module versioning is not supported, except by including a version sigil in the module name (e.g., "accounts-v1"). However, *module hashes* are a powerful feature for ensuring code safety. When a module is imported with *use*, the module hash can be specified, to tie code to a particular release.

As of Pact 2.2, `use` statements can be issued within a module declaration. This combined with module hashes provides a high level of assurance, as updated module code will fail to import if a dependent module has subsequently changed on the chain; this will also propagate changes to the loaded modules' hash, protecting downstream modules from inadvertent changes on update.

Module names must be unique within a namespace.

Interface Declaration

Interfaces contain an API specification and data definitions for smart contracts. They are comprised of:

- *function* specifications (i.e. function signatures)
- *constant* values
- *schema* definitions
- *pact* specifications
- models
- *capabilities* specifications
- *imports*

Interfaces represent an abstract api that a *module* may implement by issuing an `implements` statement within the module declaration. Interfaces may import definitions from other modules by issuing a *use* declaration, which may be used to construct new constant definitions, or make use of types defined in the imported module. Unlike Modules, Interface versioning is not supported. However, modules may implement multiple interfaces.

Interface names must be unique within a namespace.

Table Creation

Tables are created at the same time as modules. While tables are *defined* in modules, they are *created* "after" modules, so that the module may be redefined later without having to necessarily re-create the table.

The relationship of modules to tables is important, as described in *Table Guards*.

There is no restriction on how many tables may be created. Table names are namespaced with the module name.

Tables can be typed with a *schema*.

3.1.2 Transaction Execution

"Transactions" refer to business events enacted on the blockchain, like a payment, a sale, or a workflow step of a complex contractual agreement. A transaction is generally a single call to a module function. However there is no limit on how many statements can be executed. Indeed, the difference between "transactions" and "smart contract definition" is simply the *kind* of code executed, not any actual difference in the code evaluation.

3.1.3 Queries and Local Execution

Querying data is generally not a business event, and can involve data payloads that could impact performance, so querying is carried out as a *local execution* on the node receiving the message. Historical queries use a *transaction ID* as a point of reference, to avoid any race conditions and allow asynchronous query execution.

Transactional vs local execution is accomplished by targeting different API endpoints; pact code has no ability to distinguish between transactional and local execution.

3.2 Database Interaction

Pact presents a database metaphor reflecting the unique requirements of blockchain execution, which can be adapted to run on different back-ends.

3.2.1 Atomic execution

A single message sent into the blockchain to be evaluated by Pact is *atomic*: the transaction succeeds as a unit, or does not succeed at all, known as "transactions" in database literature. There is no explicit support for rollback handling, except in *multi-step transactions*.

3.2.2 Key-Row Model

Blockchain execution can be likened to OLTP (online transaction processing) database workloads, which favor denormalized data written to a single table. Pact's data-access API reflects this by presenting a *key-row* model, where a row of column values is accessed by a single key.

As a result, Pact does not support *joining* tables, which is more suited for an OLAP (online analytical processing) database, populated from exports from the Pact database. This does not mean Pact cannot *record* transactions using relational techniques – for example, a Customer table whose keys are used in a Sales table would involve the code looking up the Customer record before writing to the Sales table.

3.2.3 Queries and Performance

As of Pact 2.3, Pact offers a powerful query mechanism for selecting multiple rows from a table. While visually similar to SQL, the select and where operations offer a *streaming interface* to a table, where the user provides filter functions, and then operates on the rowset as a list data structure using sort and other functions.

```
;; the following selects Programmers with salaries >= 90000 and sorts by age␣
↪descending

(reverse (sort ['age]
  (select 'employees ['first-name,'last-name,'age]
    (and? (where 'title (= "Programmer"))
          (where 'salary (< 90000))))))

;; the same query could be performed on a list with 'filter':

(reverse (sort ['age]
  (filter (and? (where 'title (= "Programmer"))
                (where 'salary (< 90000)))
          employees)))
```

In a transactional setting, Pact database interactions are optimized for single-row reads and writes, meaning such queries can be slow and prohibitively expensive computationally. However, using the *local* execution capability, Pact can utilize the user filter functions on the streaming results, offering excellent performance.

The best practice is therefore to use select operations via local, non-transactional operations, and avoid using select on large tables in the transactional setting.

3.2.4 No Nulls

Pact has no concept of a NULL value in its database metaphor. The main function for computing on database results, with-read, will error if any column value is not found. Authors must ensure that values are present for any transactional read. This is a safety feature to ensure *totality* and avoid needless, unsafe control-flow surrounding null values.

3.2.5 Versioned History

The key-row model is augmented by every change to column values being versioned by transaction ID. For example, a table with three columns “name”, “age”, and “role” might update “name” in transaction #1, and “age” and “role” in transaction #2. Retrieving historical data will return just the change to “name” under transaction 1, and the change to “age” and “role” in transaction #2.

3.2.6 Back-ends

Pact guarantees identical, correct execution at the smart-contract layer within the blockchain. As a result, the backing store need not be identical on different consensus nodes. Pact’s implementation allows for integration of industrial RDBMSs, to assist large migrations onto a blockchain-based system, by facilitating bulk replication of data to downstream systems.

3.3 Types and Schemas

With Pact 2.0, Pact gains explicit type specification, albeit optional. Pact 1.0 code without types still functions as before, and writing code without types is attractive for rapid prototyping.

Schemas provide the main impetus for types. A schema *is defined* with a list of columns that can have types (although this is also not required). Tables are then *defined* with a particular schema (again, optional).

Note that schemas also can be used on/specified for object types.

3.3.1 Runtime Type enforcement

Any types declared in code are enforced at runtime. For table schemas, this means any write to a table will be typechecked against the schema. Otherwise, if a type specification is encountered, the runtime enforces the type when the expression is evaluated.

3.3.2 Static Type Inference on Modules

With the typecheck repl command, the Pact interpreter will analyze a module and attempt to infer types on every variable, function application or const definition. Using this in project repl scripts is helpful to aid the developer in adding "just enough types" to make the typecheck succeed. Successful typechecking is usually a matter of providing schemas for all tables, and argument types for ancillary functions that call ambiguous or overloaded native functions.

3.3.3 Formal Verification

Pact's typechecker is designed to output a fully typechecked and inlined AST for generating formal proofs in the SMT-LIB2 language. If the typecheck does not succeed, the module is not considered "provable".

We see, then, that Pact code can move its way up a "safety" gradient, starting with no types, then with "enough" types, and lastly, with formal proofs.

Note that as of Pact 2.0 the formal verification function is still under development.

3.4 Keysets and Authorization

Pact is inspired by Bitcoin scripts to incorporate public-key authorization directly into smart contract execution and administration. Pact seeks to take this further by making single- and multi-sig interactions ubiquitous and effortless with the concept of *keysets*, meaning that single-signature mode is never assumed: anywhere public-key signatures are used, single-sig and multi-sig can interoperate effortlessly. Finally, all crypto is handled by the Pact runtime to ensure programmers can't make mistakes "writing their own crypto".

Also see *Guards and Capabilities* below for how Pact moves beyond just keyset-based authorization.

3.4.1 Keyset definition

Keysets are defined by reading definitions from the message payload. Keysets consist of a list of public keys and a *keyset predicate*.

Examples of valid keyset JSON productions:

```
/* examples of valid keysets */
{
  "fully-specified-with-native-pred":
    { "keys": ["abc6bab9b88e08d","fe04ddd404feac2"], "pred": "keys-2" },

  "fully-specified-with-qual-custom":
```

(continues on next page)

(continued from previous page)

```
    { "keys": ["abc6bab9b88e08d","fe04ddd404feac2"], "pred": "my-module.custom-pred" }
↪,

  "keysonly":
    { "keys": ["abc6bab9b88e08d","fe04ddd404feac2"] }, /* defaults to "keys-all" pred␣
↪*/

  "keylist": ["abc6bab9b88e08d","fe04ddd404feac2"] /* makes a "keys-all" pred keyset␣
↪*/
}
```

3.4.2 Keyset Predicates

A keyset predicate references a function by its (optionally qualified) name, and will compare the public keys in the keyset to the key or keys used to sign the blockchain message. The function accepts two arguments, "count" and "matched", where "count" is the number of keys in the keyset and "matched" is how many keys on the message signature matched a keyset key.

Support for multiple signatures is the responsibility of the blockchain layer, and is a powerful feature for Bitcoin-style "multisig" contracts (i.e. requiring at least two signatures to release funds).

Pact comes with built-in keyset predicates: keys-all, keys-any, keys-2. Module authors are free to define additional predicates.

If a keyset predicate is not specified, keys-all is used by default.

3.4.3 Key rotation

Keysets can be rotated, but only by messages authorized against the current keyset definition and predicate. Once authorized, the keyset can be easily *redefined*.

3.4.4 Module Table Guards

When creating a table, a module name must also be specified. By this mechanism, tables are "guarded" or "encapsulated" by the module, such that direct access to the table via data-access functions is authorized only by the module's governance. However, *within module functions*, table access is unconstrained. This gives contract authors great flexibility in designing data access, and is intended to enshrine the module as the main "user data access API".

See also *module guards* for how this concept can be leveraged to protect more than just tables.

Note that as of Pact 3.5, the option has been added to selectively allow unguarded reads and transaction history access in local mode only, at the discretion of the node operator.

3.4.5 Row-level keysets

Keysets can be stored as a column value in a row, allowing for *row-level* authorization. The following code indicates how this might be achieved:

```
(defun create-account (id)
  (insert accounts id { "balance": 0.0, "keyset": (read-keyset "owner-keyset") }))

(defun read-balance (id)
```

(continues on next page)

(continued from previous page)

```
  (with-read accounts id { "balance":= bal, "keyset":= ks }
    (enforce-keyset ks)
    (format "Your balance is {}" [bal])))
```

In the example, `create-account` reads a keyset definition from the message payload using read-keyset to store as "keyset" in the table. `read-balance` only allows that owner's keyset to read the balance, by first enforcing the keyset using enforce-keyset.

3.5 Namespaces

Namespaces are defined by specifying a namespace name and associating a keyset with the namespace. Namespace scope is entered by declaring the namespace environment. All definitions issued after the namespace scope is entered will be accessible by their fully qualified names. These names are of the form *namespace.module.definition*. This form can also be used to access code outside of the current namespace for the purpose of importing module code, or implementing modules:

```
(implements my-namespace.my-interface)
;; or
(use my-namespace.my-module)
```

Code may be appended to the namespace by simply re-entering the namespace and declaring new code definitions. All definitions *must* occur within a namespace, as the global namespace (the empty namespace) is reserved for Kadena code.

Examples of valid namespace definition and scoping:

3.5.1 Example: Defining a namespace

Defining a namespace requires a keyset, and a namespace name of type string:

```
(define-keyset 'my-keyset)
(define-namespace 'my-namespace (read-keyset 'my-keyset))

pact> (namespace 'my-namespace)
"Namespace set to my-namespace"
```

3.5.2 Example: Accessing members of a namespace

Members of a namespace may be accessed by their fully-qualified names:

```
pact> (my-namespace.my-module.hello-number 3)
"Hello, your number is 3!"

;; alternatively
pact> (use my-namespace.my-module)
"Using my-namespace.my-module"
pact> (hello-number 3)
"Hello, your number is 3!"
```

3.5.3 Example: Importing module code or implementing interfaces at a namespace

Modules may be imported at a namespace, and interfaces my be implemented in a similar way. This allows the user to work with members of a namespace in a much less verbose and cumbersome way.

```
; in my-namespace
(module my-module EXAMPLE_GUARD
  (implements my-other-namespace.my-interface)

  (defcap EXAMPLE_GUARD ()
    (enforce-keyset 'my-keyset))

  (defun hello-number:string (number:integer)
    (format "Hello, your number is {}!" [number]))
)
```

3.5.4 Example: appending code to a namespace

If one is simply appending code to an existing namespace, then the namespace prefix in the fully qualified name may be ommitted, as using a namespace works in a similar way to importing a module: all toplevel definitions within a namespace are brought into scope when `(namespace 'my-namespace)` is declared. Continuing from the previous example:

```
pact> (my-other-namespace.my-other-module.more-hello 3)
"Hello, your number is 3! And more hello!"

; alternatively
pact> (namespace 'my-other-namespace)
"Namespace set to my-other-namespace"

pact> (use my-other-module)
"Using my-other-module"

pact> (more-hello 3)
"Hello, your number is 3! And more hello!"
```

3.6 Guards, Capabilities and Events

Pact 3.0 introduces powerful new concepts to allow programmers to express and implement authorization schemes correctly and easily: *guards*, which generalize keysets, and *capabilities*, which generalize authorizations or rights. In Pact 3.7, capabilities also function as *events*.

3.6.1 Guards

A guard is essentially a predicate function over some environment that enables a pass-fail operation, `enforce-guard`, to be able to test a rich diversity of conditions.

A keyset is the quintessential guard: it specifies a list of keys, and a predicate function to verify how many keys were used to sign the current transaction. Enforcement happens via `enforce-keyset`, causing the transaction to fail if the necessary keys are not found in the signing set.

However, there are other predicates that are equally useful:

- We might want to enforce that a *module* is the only entity that can perform some function, for instance to debit some account.
- We might want to ensure that a user has provided some secret, like a hash preimage, as seen in atomic swaps.
- We might want to combine all of the above into a single, enforceable rule: "ensure user A signed the transaction AND provided a hash preimage AND is only executable by module `foo`".

Finally, we want guards to *interoperate* with each other, so that smart contract code doesn't have to worry about what kind of guard is used to mediate access to some resource or right. For instance, it is easy to think of entries in a ledger having diverse guards, where some tokens are guarded by keysets, while others are autonomously owned by modules, while others are locked in some kind of escrow transaction: what's important is that the guard always be enforced for the given account, not what type of guard it is.

Guards address all of these needs. Keysets are now just one type of guard, to which we add module guards, pact guards, and completely customizable "user guards". You can store any type of guard in the database using the `guard` type. The `keyset` type is still supported, but developers should switch to `guard` to enjoy the enhanced flexibility.

3.6.2 Capabilities

Capabilities are a new construct in Pact 3.0 that draws from capability theory to offer a system for managing runtime user rights in an explicit, literate, and principled fashion.

Simply put, a *capability* is a "ticket" that when *acquired* allows the user to perform some sensitive task. If the user is unable to acquire the ticket, portions of the transaction that demand the ticket will fail.

Using capabilities to protect code

Code can demand that a capability be "already granted", that is, make no attempt to acquire the ticket, but fail if it was not acquired somewhere else. This is done with the construct `require-capability`.

Code can also directly attempt to acquire a capability, but only for a specific *scope*. This is done with the special form `with-capability`, which, like `with-read`, scopes a body of code. Here, the ticket is granted while this body of code is executing, and is revoked when the body leaves execution.

Expressing capabilities in code: `defcap`

We've described capabilities like a "ticket", so let's continue by adding some attributes to this ticket:

- It needs a general name, like "ALLOW_ENTRY", to identify the operation being protected.
- It needs *parameters*, so that a capability can be granted to a specific entity ("user-id"), and/or for a particular amount ("amount" some decimal, "active" flag).
- It needs a *predicate function* to perform whatever tests govern whether to grant the ticket.

Pact provides the `defcap` construct to do this.

```
(defcap ALLOW_ENTRY (user-id:string)
  "Govern entry operation."
  (with-read table user-id
    { "guard" := guard, "active" := active }
    (enforce-guard guard)
    (enforce active "Only active users allowed entry")))
```

`ALLOW_ENTRY` is the name or *domain* of the capability. `user-id` is a *parameter*. Together, they form the *specification* of a capability. Thus, `(ALLOW_ENTRY 'dave)` and `(ALLOW_ENTRY 'carol)` describe separate capabilities. (Note that capability theory's notion of *designation* is indicated here, which we'll return to when we discuss capabilities and signatures).

The body implements the predicate function. It accesses whatever data it needs to perform necessary tests to protect against improper granting of the ticket. The body can do more than that – it can import or *compose* additional capabilities, for instance – and it can even modify database state. This might be used to ensure a capability cannot be granted ever again after the first time it is acquired, for example.

To acquire this capability, you would invoke `with-capability`:

```
(defun enter (user-name)
  (with-capability (ALLOW_ENTRY user-name)
    (do-entry user-name)            ;; call "protected" function
    (update-entry-status user-name) ;; update database
  )
  (record-audit "ENTRY" user-name)  ;; some "unsafe" operation
)
```

To demand or *require* the capability, you would use `require-capability`:

```
(defun do-entry (user-name)
  (require-capability (ALLOW_ENTRY user-name))
  ...
)
```

Requiring capabilities allow for "private" or "restricted" functions than cannot be called directly. Here we see that `do-entry` can only be called "privately", by code inside the module somewhere. What's more, it can only be called in an outer operation for this user in particular, "restricting" it to that user.

Composing capabilities

A `defcap` can "import" other capabilities, for modular factoring of guard code, or to "compose" the outer capability from "smaller", "inner" capabilities.

```
(defcap ALLOW_ENTRY (user-id:string)
  "Govern entry operation."
  (with-read table user-id
    { "guard" := guard, "active" := active }
    (enforce-guard guard)
    (enforce active "Only active users allowed entry")
    (compose-capability DB_LOG) ;; allow db logging while ALLOW_ENTRY is in scope
    ))
```

Composed capabilities are only in scope when their "parent" capability is granted.

3.6.3 Signature capabilities

In Pact transaction messages, each signer can "scope" their signature to one or more capabilities. This restricts keyset guard operations on that signature: keysets demanding the scoped signature will only succeed while the ticket is held, or is in the process of being acquired – keysets are often checked in order to grant a capability.

This "scoping" allows the signer to safely call untrusted code. For instance, in the Chainweb gas system, the "sender" signs the message to fund whatever gas costs are charged for the transaction. By signing the message, the sender has potentially allowed any code to debit from their account!

With that sender's signature has `(GAS)` added to it, it is scoped within gas payments in the coin contract only. Third-party code is prohibited from accessing that account during the transaction.

3.6.4 Signatures and Managed Capabilities

Signature capabilities are also a mechanism to *install* capabilities, but only if that capability is *managed*. "Vanilla" capabilities are just tickets to show before you try some protected operation, but *managed* capabilities are able to *change the state* of a capability as it is brought into and out of scope. The ticket metaphor breaks down here, as this is now a dynamic object that mediates whether capabilities are acquired.

If a signer attaches a managed capability to their signature list, the capability is "installed", which is not the same as "granted" or "acquired": if the capability's predicate function allows this signer to install the capability, the installed version will then govern any code needing the capability to unlock some protected operation, by means of a *manager function*.

Capability management with a manager function

A managed capability allows for safe interoperation with otherwise untrusted code. By signing with a managed capability, you are *allowing* some untrusted code to *request* grant of the capability; if the capability was not in the signature list, the untrusted code cannot request it.

If the capability *manager function* doesn't grant the request, the untrusted code fails to execute. The common usage of this is to grant a payment to third-party code, such that the third-party code can directly transfer on behalf of the user some amount of coin, but only up to the indicated amount.

The TRANSFER managed capability

```
(defcap TRANSFER (sender:string receiver:string amount:decimal)
  @managed amount TRANSFER_mgr
  (compose-capability (DEBIT sender))
  (compose-capability (CREDIT receiver)))

(defun TRANSFER_mgr:decimal (managed:decimal requested:decimal)
  (enforce (>= managed requested) "Transfer quantity exhausted")
  (- managed requested) ;; update managed quantity for next time
)
```

`TRANSFER` allows for `sender` to approve any number of payments to `receiver` up to some `amount`. Once the amount is exceeded, the capability can no longer be brought into scope.

This allows third-party code to directly enact payments. Managed capabilities are an important feature to allow smart contracts to directly call some other trusted code in a tightly-constrained context.

Automatic "one-shot" capability management

A managed capability that does not specify a manager function is "auto-managed", meaning that after install, the capability can be granted exactly once for the given parameters. Further attempts will fail after the initial grant goes out of scope.

In the following example, the capability will have "one-shot" automatic management:

```
(defcap VOTE (member:string)
  @managed
  (validate-member member))
```

3.6.5 Guards vs Capabilities

Guards and capabilities can be confusing: given we have guards like keysets, what do we need the capability concept for?

Guards allow us to define a *rule* that must be satisfied for the transaction to proceed. As such, they really are just a way to declare a pass-fail condition or predicate. The Pact guard system is flexible enough to express any rule you can code.

Capabilities allow us to declare how that rule is deployed to grant some authority. In doing so, they enumerate the critical rights that are extended to users of the smart contract, and "protect" code from being called incorrectly.

Note also that **capabilities can only be granted inside the module code that declares them**, whereas guards are simply data that can be tested anywhere. This is an important security property, as it ensures an attacker cannot elevate their privileges from outside the module code.

3.6.6 Modeling capabilities with `compose-capability`

The only problem with the above code is it pushed the awareness of DEBIT into the `transfer` function, whereas separation of concerns would better have it housed in `debit`. What's more, we'd like to ensure that `debit` is always called in a "transfer" capacity, that is, that the corresponding `credit` occurs. Thus, the better way to model this is with two capabilities, with TRANSFER being a "no-guard" capability that simply encloses `debit` and `credit` calls:

```
(defcap TRANSFER (from to amount)
  (compose-capability (DEBIT from))
  (compose-capability (CREDIT to)))

(defcap DEBIT (from)
  (enforce-guard (at 'guard (read table from))))

(defcap CREDIT (to)
  (check-account-exists to))

(defun transfer (from to amount)
  (with-capability (TRANSFER to from amount)
    (debit from amount)
    (credit to amount)))

(defun debit (user amount)
  (require-capability (DEBIT user))
    (update accounts user ...))

(defun credit (user amount)
  (require-capability (CREDIT user)
    (update accounts user ...)))
```

Thus, `TRANSFER` protects `debit` and `credit` from being used independently, while `DEBIT` governs specifically the ability to debit, enforcing the guard, while `CREDIT` simply creates a "restricted" capability for `credit`.

3.6.7 Improving efficiency

Once capabilities are granted they are installed into the pact environment for the scope of the call to `with-capability`; once that form is exited, the capability is uninstalled. This scoping prevents duplicate testing of the predicate: **capabilities that have already been acquired (or installed) and are in-scope are not re-evaluated**, either by acquiring or requiring.

3.6.8 `defcap` details

Since a `defcap` production both *specifies* a "domain" of capability instances, and *implements* the guard function, it has some surprising features. Since capability grant is cached in the environment, the function does not need to be called when invoked in `with-capability` or `require-capability` asks for some already-granted ticket.

As a result, **"defcap"s cannot be executed directly**, as arbitrary execution would violate the semantics described here. This is an important security property as it ensures that the granting code can only be called in approved contexts, inside the module.

3.6.9 Testing scoping signatures with capabilities

Scoped signatures can be tested using the new `env-sigs` REPL function as follows:

```
(module accounts GOV
  ...
  (defcap PAY (sender receiver amount)
    (enforce-keyset (at 'keyset (read accounts sender))))

  (defun pay (sender receiver amount)
    (with-capability (PAY sender receiver amount)
      (transfer sender receiver amount)))
  ...
)

(set-sigs [{'key: "alice", 'caps: ["(accounts.PAY \"alice\" \"bob\" 10.0)"]}])
(accounts.pay "alice" "bob" 10.0) ;; works as the cap match the signature caps

(set-sigs [('key: "alice", 'caps: ["(accounts.PAY \"alice\" "\carol\" 10.0)"]}])
(expect-failure "payment to bob will no longer be able to enforce alice's keyset"
  (accounts.pay "alice" "bob" 10.0))
```

3.6.10 Guard types

Guards come in five flavors: keyset, keyset reference, module, pact, and user guards.

Keyset guards.

These are the classic pact keysets. Using the `keyset` type is the one instance where you can restrict a guard subtype, otherwise the `guard` type obscures the implementation type to prevent developers from engaging in guard-specific control flow, which would be against best practices. Again, it is better to switch to `guard` unless there is a specific need to use keysets.

```
(enforce-guard (read-keyset "keyset"))
```

Keyset reference guards

Keysets can be installed into the environment with `define-keyset`, but if you wanted to store a reference to a defined keyset, you would need to use a `string` type. To make environment keysets interoperate with concrete keysets and other guards, we introduce the "keyset reference guard" which indicates that a defined keyset is used instead of a concrete keyset.

```
(enforce-guard (keyset-ref-guard "foo"))

(update accounts user { "guard": (keyset-ref-guard "foo") })
```

Module guards

Module guards are a special guard that when enforced will fail unless:

- the code calling the enforce was called from within the module, or
- module governance is granted to the current transaction.

This is for allowing a module or smart contract to autonomously "own" and manage some asset. As such it is operationally identical to how module table access is guarded: only module code or a transaction having module admin can directly write to a module tables, or upgrade the module, so there is no need to use a module guard for these in-module operations. A module guard is used to "project" module admin outside of the module (e.g. to own coins in an external ledger), or "inject" module admin into an internal database representation (e.g. to own an internally-managed asset alongside other non-module owners).

See *Module Governance* for more information about module admin management.

`create-module-guard` takes a `string` argument to allow naming the guard, to indicate the purpose or role of the guard.

```
(enforce-guard (create-module-guard "module-owned-asset"))
```

Pact guards

Pact guards are a special guard that will only pass if called in the specific `defpact` execution in which the guard was created.

Imagine an escrow transaction where the funds need to be moved into an escrow account: if modeled as a two-step pact, the funds can go into a special account named after the pact id, guarded by a pact guard. This means that only code in a subsequent step of that particular pact execution (ie having the same pact ID) can pass the guard.

```
(defpact escrow (from to amount)
  (step (with-capability (ESCROW) (init-escrow from amount)))
  (step (with-capability (ESCROW) (complete-escrow to amount))))

(defun init-escrow (from amount)
  (require-capability (ESCROW))
  (create-account (pact-id) (create-pact-guard "escrow"))
  (transfer from (pact-id) amount))

(defun complete-escrow (to amount)
  (require-capability (ESCROW))
  (with-capability (USER_GUARD (pact-id)) ;; enforces guard on account (pact-id)
    (transfer (pact-id) to amount)))
```

Pact guards turn pact executions into autonomous processes that can own assets, and is a powerful technique for trustless asset management within a multi-step operation.

User guards

User guards allow the user to design an arbitrary predicate function to enforce the guard, given some initial data. For instance, a user guard could be designed to require two separate keysets to be enforced:

```
(defun both-sign (ks1 ks2)
  (enforce-keyset ks1)
  (enforce-keyset ks2))

(defun install-both-guard ()
  (write guard-table "both"
    { "guard":
      (create-user-guard
        (both-sign (read-keyset "ks1) (read-keyset "ks2")))
    }))


(defun enforce-both-guard ()
  (enforce-guard (at "guard" (read guard-table "both"))))
```

User guards can seem similar to capabilities but are different, namely in that they can be stored in the database and passed around like plain data. Capabilities are in-module rights that can only be enforced within the declaring module, and offer scoping and the other benefits mentioned above. User guards are for implementing custom predicate logic that can't be expressed by other built-in guard types.

HTLC guard example

The following example shows how a "hash timelock" guard can be made, to implement atomic swaps.

```
(create-hashlock-guard (secret-hash timeout signer-ks)
  (create-user-guard (enforce-hashlock secret-hash timeout signer-ks)))

(defun enforce-hashlock (secret-hash timeout signer-ks)
  (enforce-one [
    (enforce (= (hash (read-msg "secret")) secret-hash))
    (and
      (enforce-keyset signer-ks)
      (enforce (> (at "block-time" (chain-data)) timeout) "Timeout not passed"))
      ]))
```

3.6.11 Events

Pact 3.7 introduces *events* which are emitted in the course of a transaction and included in the transaction receipt to allow for monitoring and proving via SPV that a particular event transpired.

In Pact, events are modeled as capabilities, for the following reasons: - Capabilities already have the right shape for an event, which is essentially arbitrary data published under a topic or name. With capabilities, the capability name is the topic, and the arguments are the data. - The acquisition of managed capabilities are a bona-fide event. Events complete the managed lifecycle, where you might install/approve a capability of some quantity on the way in, but not necessarily see what quantity was used. With events, the output of the actually acquired capability is present in the receipt. - Capabilities are protected such that they can only be acquired in module code, which is appropriate as well for events.

The @event metadata tag

Any capability can cause events to be emitted upon acquisition by using the `@event` metadata tag.

```
(defcap BURN(qty:decimal)
  @event
  ...
)
```

`@event` cannot be used alongside `@managed`, because ...

Managed capabilities are automatically eventing

Managed capabilites emit events automatically with the parameters specified in acquisition (as opposed to install). From an eventing point of view, managed capabilities are those capabilities that can only "happen once". Whereas, a non-managed, eventing capability can fire events an arbitrary amount of times.

Testing for events

Use *env-events* to test for emitted events in repl scripts.

3.7 Generalized Module Governance

Before Pact 3.0, module upgrade and administration was governed by a defined keyset that is referenced in the module definition. With Pact 3.0, this `string` value can alternately be an unqualified bareword that references a `defcap` within the module body. This `defcap` is the *module governance capability*.

With the introduction of the governance capability syntax, Pact modules now support *generalized module governance*, allowing for module authors to design any governance scheme they wish. Examples include tallying a stakeholder vote on an upgrade hash, or enforcing more than one keyset.

3.7.1 Keysets vs governance functions

To illustrate, let's consider a module governed by a keyset:

```
(module foo 'foo-keyset ...)
```

This indicates that if a user tried to upgrade the module, or directly write to the module tables, `'foo-keyset` would be enforced on the transaction signature set.

This can be directly implemented in a governance capability as follows:

```
(module foo GOVERNANCE
  ...
  (defcap GOVERNANCE ()
    (enforce-keyset 'foo-keyset))
  ...
)
```

Note the capability can have whatever name desired; GOVERNANCE is a good idiomatic name however.

3.7.2 Governance capability and module admin

As a `defcap`, the governance function cannot be called directly by user code. It is automatically invoked in the following circumstances:

- A module upgrade is being attempted
- Module tables are being directly accessed outside the module code
- A *module guard* for this module is being enforced.

In these cases, the transaction is tested for elevated access to "module admin", defined as the grant of the *module admin capability*. This capability cannot be expressed in user code, so it cannot be installed, acquired, required or composed.

However, the implementing capability, here called `GOVERNANCE`, can be installed or acquired etc. If passed, this gets scoped like any normal capability, here over some protected code that only module admins can run.

Module admin capability scope

The special module admin capability, once automatically installed in the cases described above, **stays in scope for the rest of the calling transaction**. This is unlike "user" capabilities, which can only be acquired in a fixed scope specified by the body of `with-capability`.

This may sound worrisome, but the rationale is that a governance capability once granted should not be based on some transient fact that can become false during a single transaction. This is important especially in module upgrades, *which can change the governance capability itself*: if the module admin was tested again this could cause the upgrade to fail, for instance when migrating data with direct table rights.

Capability risks

Also, this means that, when initially installing a module, *the governance function is not invoked*. This is different behavior than when a keyset is specified: the keyset must be defined and it is enforced, to ensure that the keyset actually exists.

Module governance is therefore more "risky" as it can mean that the module cannot be upgraded if there is a bug in the governance capability. Clearly, care must be taken when implementing module capabilities, and using the Pact formal verification system is highly recommended here.

3.7.3 Example: stakeholder upgrade vote

In the following code, a module can be upgraded based on a vote. An upgrade is designed as a Pact transaction, and its hash and code are distributed to stakeholders, who vote for the upgrade. Once the upgrade is sent in, the vote is tallied in the governance capability, and if a simple majority is found, the code is upgraded.

```
(module govtest count-votes
  "Demonstrate programmable governance showing votes \
 \ for upgrade transaction hashes"
  (defschema vote
    vote-hash:string)

  (deftable votes:{vote})

  (defun vote-for-hash (user hsh)
    "Register a vote for a particular transaction hash"
```

(continues on next page)

(continued from previous page)

```
    (write votes user { "vote-hash": hsh })
  )

  (defcap count-votes ()
    "Governance capability to tally votes for the upgrade hash".
    (let* ((h (tx-hash))
           (tally (fold (do-count h)
                        { "for": 0, "against": 0 }
                        (keys votes)))
          )
      (enforce (> (at 'for tally) (at 'against tally))
               (format "vote result: {}, {}" [h tally]))))
  )

  (defun do-count (hsh tally u)
    "Add to TALLY if U has voted for HSH"
    (bind tally { "for" := f, "against" := a }
      (with-read votes u { 'vote-hash := v }
        (if (= v hsh)
            { "for": (+ 1 f), "against": a }
          { "for": f, "against": (+ 1 a) }))))
  )
```

3.8 Interfaces

An interface, as defined in Pact, is a collection of models used for formal verification, constant definitions, and typed function signatures. When a module issues an *implements*, then that module is said to 'implement' said interface, and must provide an implementation . This allows for abstraction in a similar sense to Java's interfaces, Scala's traits, Haskell's typeclasses or OCaML's signatures. Multiple interfaces may be implemented in a given module, allowing for an expressive layering of behaviors.

Interfaces are declared using the `interface` keyword, and providing a name for the interface. Since interfaces cannot be upgraded, and no function implementations exist in an interface aside from constant data, there is no notion of governance that need be applied. Multiple interfaces may be implemented by a single module. If there are conflicting function names among multiple interfaces, then the two interfaces are incompatible, and the user must either inline the code they want, or redefine the interfaces to the point that the conflict is resolved.

Constants declared in an interface can be accessed directly by their fully qualified name `namespace.interface.const`, and so, they do not have the same naming constraints as function signatures.

Additionally, interfaces my make use of module declarations, admitting use of the use keyword, allowing interfaces to import members of other modules. This allows interface signatures to be defined in terms of table types defined in an imported module.

3.8.1 Example: Declaring and implementing an interface

```
(interface my-interface
    (defun hello-number:string (number:integer)
      @doc "Return the string \"Hello, $number!\" when given a string"
        )

    (defconst SOME_CONSTANT 3)
```

(continues on next page)

(continued from previous page)

```
)

(module my-module (read-keyset 'my-keyset)
    (implements my-interface)

    (defun hello-number:string (number:integer)
        (format "Hello, {}!" [number]))

    (defun square-three ()
        (* my-interface.SOME_CONSTANT my-interface.SOME_CONSTANT))
)
```

3.8.2 Declaring models in an interface

Formal verification is implemented at multiple levels within an interface in order to provide an extra level of security. Models may be declared either within the body of the interface or at the function level in the same way that one would declare them in a module, with the exception that not all models are applicable to an interface. Indeed, since there is no abstract notion of tables for interfaces, abstract table invariants cannot be declared. However, if an interface imports table schema and types from a module via the use keyword, then the interface can define body and function models that apply directly to the concrete table type. Otherwise, all properties are candidates for declaration in an interface.

When models are declared in an interface, they are appeneded to the list of models present in the implementing module at the level of declaration: body-level models are appended to body-level models, and function-level models are appended to function-level models. This allows users to extend the constraints of an interface with models applicable to specific business logic and implementation.

Declaring models shares the same syntax with modules:

Example: declaring models, tables, and importing modules in an interface

```
(interface coin-sig

  "Coin Contract Abstract Interface Example"

  (use acct-module)

  (defun transfer:string (from:string to:string amount:integer)
    @doc   "Transfer money between accounts"
    @model [(property (row-enforced accounts "ks" from))
            (property (> amount 0))
            (property (= 0 (column-delta accounts "balance")))
            ]
  )
)
```

3.9 Module References

Pact 3.7 gains a form of *genericism* with *module references*. This is motivated by the desire to interoperate between modules that implement a common interface, and to be able to treat the indicated module as a data value to gain *polymorphism* across modules.

Modules and interfaces thus need to be referenced directly, which is simply accomplished by issuing their name in code.

```
(module foo 'k
  (defun bar () 0))

(namespace ns)

(interface bar
  (defun quux:string ()))

(module zzz 'k
  (implements bar)
  (defun quux:string () "zzz"))

foo ;; module reference to 'foo', of type 'module'
ns.bar ;; module reference to `bar` interface, also of type 'module'
ns.zzz ;; module reference to `zzz` module, of type 'module{ns.bar}'
```

Using a module reference in a function is accomplished by specifying the type of the module reference argument, and using the *dereference operator* `::` to invoke a member function of the interfaces specified in the type.

```
(interface baz
  (defun quux:bool (a:integer b:string))
  (defconst ONE 1)
  )
(module impl 'k
  (implements baz)
  (defun quux:bool (a:integer b:string)
    (> (length b) a))
  )

...

(defun foo (bar:module{baz})
  (bar::quux 1 "hi") ;; derefs 'quux' on whatever module is passed in
  bar::ONE             ;; directly references interface const
)

...

(foo impl) ;; 'impl' references the module defined above, of type 'module{baz}'
```

Module references can be used as normal pact values, which includes storage in the database.

3.10 Computational Model

Here we cover various aspects of Pact's approach to computation.

3.10.1 Turing-Incomplete

Pact is turing-incomplete, in that there is no recursion (recursion is detected before execution and results in an error) and no ability to loop indefinitely. Pact does support operation on list structures via map, fold and filter, but since there is no ability to define infinite lists, these are necessarily bounded.

Turing-incompleteness allows Pact module loading to resolve all references in advance, meaning that instead of addressing functions in a lookup table, the function definition is directly injected (or "inlined") into the callsite. This is an example of the performance advantages of a Turing-incomplete language.

3.10.2 Single-assignment Variables

Pact allows variable declarations in *let expressions* and *bindings*. Variables are immutable: they cannot be re-assigned, or modified in-place.

A common variable declaration occurs in the with-read function, assigning variables to column values by name. The bind function offers this same functionality for objects.

Module-global constant values can be declared with *defconst*.

3.10.3 Data Types

Pact code can be explicitly typed, and is always strongly-typed under the hood as the native functions perform strict type checking as indicated in their documented type signatures.

Pact's supported types are:

- *Strings*
- *Integers*
- *Decimals*
- *Booleans*
- *Time values*
- *Keysets* and *Guards*
- *Lists*
- *Objects*
- *Function*, *pact*, and *capability* definitions
- *Tables*
- *Schemas*

3.10.4 Performance

Pact is designed to maximize the performance of *transaction execution*, penalizing queries and module definition in favor of fast recording of business events on the blockchain. Some tips for fast execution are:

Single-function transactions

Design transactions so they can be executed with a single function call.

Call with references instead of `use`

When calling module functions in transactions, use *reference syntax* instead of importing the module with *use*. When defining modules that reference other module functions, `use` is fine, as those references will be inlined at module definition time.

Hardcoded arguments vs. message values

A transaction can encode values directly into the transactional code:

```
(accounts.transfer "Acct1" "Acct2" 100.00)
```

or it can read values from the message JSON payload:

```
(defun transfer-msg ()
  (transfer (read-msg "from") (read-msg "to")
            (read-decimal "amount")))
...
(accounts.transfer-msg)
```

The latter will execute slightly faster, as there is less code to interpret at transaction time.

Types as necessary

With table schemas, Pact will be strongly typed for most use cases, but functions that do not use the database might still need types. Use the typecheck REPL function to add the necessary types. There is a small cost for type enforcement at runtime, and too many type signatures can harm readability. However types can help document an API, so this is a judgement call.

3.10.5 Control Flow

Pact supports conditionals via if, bounded looping, and of course function application.

Use enforce

"If" should never be used to enforce business logic invariants: instead, enforce is the right choice, which will fail the transaction.

Indeed, failure is the only *non-local exit* allowed by Pact. This reflects Pact's emphasis on *totality*.

Note that enforce-one (added in Pact 2.3) allows for testing a list of enforcements such that if any pass, the whole expression passes. This is the sole example in Pact of "exception catching" in that a failed enforcement simply results in the next test being executed, short-circuiting on success.

Use built-in keyset predicates

The built-in keyset functions keys-all, keys-any, keys-2 are hardcoded in the interpreter to execute quickly. Custom keysets require runtime resolution which is slower.

3.10.6 Functional Concepts

Pact includes the functional-programming "greatest hits": map, fold and filter. These all employ *partial application*, where the list item is appended onto the application arguments in order to serially execute the function.

```
(map (+ 2) [1 2 3])
(fold (+) "" ["Concatenate" " " "me"])
```

Pact also has compose, which allows "chaining" applications in a functional style.

3.10.7 Pure execution

In certain contexts Pact can guarantee that computation is "pure", which simply means that the database state will not be modified. Currently, `enforce`, `enforce-one` and keyset predicate evaluation are all executed in a pure context. *defconst* memoization is also pure.

3.10.8 LISP

Pact's use of LISP syntax is intended to make the code reflect its runtime representation directly, allowing contract authors focus directly on program execution. Pact code is stored in human-readable form on the ledger, such that the code can be directly verified, but the use of LISP-style *s-expression syntax* allows this code to execute quickly.

3.10.9 Message Data

Pact expects code to arrive in a message with a JSON payload and signatures. Message data is read using read-msg and related functions. While signatures are not directly readable or writable, they are evaluated as part of *keyset predicate* enforcement.

JSON support

Values returned from Pact transactions are expected to be directly represented as JSON values.

When reading values from a message via read-msg, Pact coerces JSON types as follows:

- String -> `string`
- Number -> `decimal`
- Boolean -> `bool`
- Object -> `object`
- Array -> `list`

Integer values are represented as objects and read using read-integer.

3.11 Confidentiality

Pact is designed to be used in a *confidentiality-preserving* environment, where messages are only visible to a subset of participants. This has significant implications for smart contract execution.

3.11.1 Entities

An *entity* is a business participant that is able or not able to see a confidential message. An entity might be a company, a group within a company, or an individual.

3.11.2 Disjoint Databases

Pact smart contracts operate on messages organized by a blockchain, and serve to produce a database of record, containing results of transactional executions. In a confidential environment, different entities execute different transactions, meaning the resulting databases are now *disjoint*.

This does not affect Pact execution; however, database data can no longer enact a "two-sided transaction", meaning we need a new concept to handle enacting a single transaction over multiple disjoint datasets.

3.11.3 Confidential Pacts

An important feature for confidentiality in Pact is the ability to orchestrate disjoint transactions in sequence to be executed by targeted entities. This is described in the next section.

3.12 Asynchronous Transaction Automation with "Pacts"

"Pacts" are multi-stage sequential transactions that are defined as a single body of code called a *pact*. Defining a multi-step interaction as a pact ensures that transaction participants will enact an agreed sequence of operations, and offers a special "execution scope" that can be used to create and manage data resources only during the lifetime of a given multi-stage interaction.

Pacts are a form of *coroutine*, which is a function that has multiple exit and re-entry points. Pacts are composed of *steps* such that only a single step is executed in a given blockchain transaction. Steps can only be executed in strict sequential order.

A pact is defined with arguments, similarly to function definition. However, arguments values are only evaluated in the execution of the initial step, after which those values are available unchanged to subsequent steps. To share new values with subsequent steps, a step can yield values which the subsequent step can recover using the special resume binding form.

Pacts are designed to run in one of two different contexts, private and public. A private pact is indicated by each step identifying a single entity to execute the step, while public steps do not have entity indicators. A pact can only be uniformly public or private: if some steps has entity indicators and others do not, this results in an error at load time.

3.12.1 Public Pacts

Public pacts are comprised of steps that can only execute in strict sequence. Any enforcement of who can execute a step happens within the code of the step expression. All steps are "manually" initiated by some participant in the transaction with CONTINUATION commands sent into the blockchain.

3.12.2 Private Pacts

Private pacts are comprised of steps that execute in sequence where each step only executes on entity nodes as selected by the provided 'entity' argument in the step; other entity nodes "skip" the step. Private pacts are executed automatically by the blockchain platform after the initial step is sent in, with the executing entity's node automatically sending the CONTINUATION command for the next step.

3.12.3 Failures, Rollbacks and Cancels

Failure handling is dramatically different in public and private pacts.

In public pacts, a rollback expression is specified to indicate that the pact can be "cancelled" at this step with a participant sending in a CANCEL message before the next step is executed. Once the last step of a pact has been executed, the pact will be finished and cannot be rolled back. Failures in public steps are no different than a failure in a non-pact transaction: all changes are rolled back. Pacts can therefore only be canceled explicitly and should be modeled to offer all necessary cancel options.

In private pacts, the sequential execution of steps is automated by the blockchain platform itself. A failure results in a ROLLBACK message being sent from the executing entity node which will trigger any rollback expression specified in the previous step, to be executed by that step's entity. This failure will then "cascade" to the previous step as a new ROLLBACK transaction, completing when the first step is rolled back.

3.12.4 Yield and Resume

A step can yield values to the following step using yield and resume. In public, this is an unforgeable value, as it is maintained within the blockchain pact scope. In private, this is simply a value sent with a RESUME message from the executed entity.

3.12.5 Pact execution scope and `pact-id`

Every time a pact is initiated, it is given a unique ID which is retrievable using the pact-id function, which will return the ID of the currently executing pact, or fail if not running within a pact scope. This mechanism can thus be used to guard access to resources, analogous to the use of keysets and signatures. One typical use of this is to create escrow accounts that can only be used within the context of a given pact, eliminating the need for a trusted third party for many use-cases.

3.12.6 Testing pacts

Pacts can be tested in repl scripts using the env-entity, env-step and pact-state repl functions to simulate pact executions.

It is also possible to simulate pact execution in the pact server API by formatting *continuation Request* yaml files into API requests with a `cont` payload.

3.13 Dependency Management

Pact supports a number of features to manage a module's dependencies on other Pact modules.

3.13.1 Module Hashes

Once loaded, a Pact module is associated with a hash computed from the module's source code text. This module hash uniquely identifies the version of the module. Hashes are base64url-encoded BLAKE2 256-bit hashes. Module hashes can be examined with describe-module:

```
pact> (at "hash" (describe-module 'accounts))
"ZHD9IZg-ro1wbx7dXi3Fr-CVmA-Pt71Ov9M1UNhzAkY"
```

3.13.2 Pinning module versions with `use`

The *use* special form allows a module hash to be specified, in order to pin the dependency version. When used within a module declaration, it introduces the dependency hash value into the module's hash. This allows a "dependency-only" upgrade to push the upgrade to the module version.

3.13.3 Inlined Dependencies: "No Leftpad"

When a module is loaded, all references to foreign modules are resolved, and their code is directly inlined. At this point, upstream definitions are permanent: the only way to upgrade dependencies is to reload the original module.

This permanence is great for user code: once a module is loaded, an upstream provider cannot change what code is executed within. However, this creates a big problem for upstream developers, as they cannot upgrade the downstream code themselves in order to address an exploit, or to introduce new features.

3.13.4 Blessing hashes

A trade-off is needed to balance these opposing interests. Pact offers the ability for upstream code to break downstream dependent code at runtime. Table access is guarded to enforce that the module hash of the inlined dependency either matches the runtime version, or is in a set of "blessed" hashes, as specified by *bless* in the module declaration:

```
(module provider 'keyset
  (bless "ZHD9IZg-ro1wbx7dXi3Fr-CVmA-Pt71Ov9M1UNhzAkY")
  (bless "bctSHEz4N5Y1XQaic6eOoBmjty88HMMGfAdQLPuIGMw")
  ...
)
```

Dependencies with these hashes will continue to function after the module is loaded. Unrecognized hashes will cause the transaction to fail. However, "pure" code that does not access the database is unaffected. This prevents a "leftpad situation" where trivial utility functions can harm downstream code stability.

3.13.5 Phased upgrades with "v2" modules

Upstream providers can use the bless mechanism to phase in an important upgrade, by renaming the upgraded module to indicate the new version, and replacing the old module with a new, empty module that only blesses the last version (and whatever earlier versions desired). New clients will fail to import the "v1" code, requiring them to use the new version, while existing users can continue to use the old version, presumably up to some advertised time limit. The "empty" module can offer migration functions to handle migrating user data to the new module, for the user to self-upgrade in the time window.

CHAPTER 4

Syntax

4.1 Literals

4.1.1 Strings

String literals are created with double-ticks:

```
pact> "a string"
"a string"
```

Strings also support multiline by putting a backslash before and after whitespace (not interactively).

```
(defun id (a)
  "Identity function. \
  \Argument is returned."
  a)
```

4.1.2 Symbols

Symbols are string literals representing some unique item in the runtime, like a function or a table name. Their representation internally is simply a string literal so their usage is idiomatic.

Symbols are created with a preceding tick, thus they do not support whitespace nor multiline syntax.

```
pact> 'a-symbol
"a-symbol"
```

4.1.3 Integers

Integer literals are unbounded, and can be positive or negative.

```
pact> 12345
12345
pact> -92233720368547758071238746123 4
-92233720368547758071238746123 4
```

4.1.4 Decimals

Decimal literals have potentially unlimited precision.

```
pact> 100.25
100.25
pact> -356452.2345187282874610238565823829837 46
-356452.2345187282874610238565823829837 46
```

4.1.5 Booleans

Booleans are represented by `true` and `false` literals.

```
pact> (and true false)
false
```

4.1.6 Lists

List literals are created with brackets, and optionally separated with commas. Uniform literal lists are given a type in parsing.

```
pact> [1 2 3]
[1 2 3]
pact> [1,2,3]
[1 2 3]
pact> (typeof [1 2 3])
"[integer]"
pact> (typeof [1 2 true])
"list"
```

4.1.7 Objects

Objects are dictionaries, created with curly-braces specifying key-value pairs using a colon `:`. For certain applications (database updates), keys must be strings.

```
pact> { "foo": (+ 1 2), "bar": "baz" }
{ "foo": 3, "bar": "baz" }
```

4.1.8 Bindings

Bindings are dictionary-like forms, also created with curly braces, to bind database results to variables using the `:=` operator. They are used in with-read, with-default-read, bind and resume to assign variables to named columns in a row, or values in an object.

```
(defun check-balance (id)
  (with-read accounts id { "balance" := bal }
    (enforce (> bal 0) (format "Account in overdraft: {}" [bal]))))
```

4.1.9 Lambdas

Lambdas, or "anonymous functions", allow defining functions to be applied in local scope, as opposed to defining functions at top-level with `defun`.

Lambdas are supported in `let`, `let*`, and as inline arguments to built-in function applications.

```
; identity function
(let ((f (lambda (x) x))) (f a))
; native example
(let ((f (lambda (x) x))) (map (f) [1 2 3]))
; Inline native example:
(map (lambda (x) x) [1 2 3])
```

4.2 Type specifiers

Types can be specified in syntax with the colon `:` operator followed by a type literal or user type specification.

4.2.1 Type literals

- `string`
- `integer`
- `decimal`
- `bool`
- `time`
- `keyset`
- `list`, or `[type]` to specify the list type
- `object`, which can be further typed with a schema
- `table`, which can be further typed with a schema
- `module`, which must be further typed with required interfaces.

4.2.2 Schema type literals

A schema defined with *defschema* is referenced by name enclosed in curly braces.

```
table:{accounts}
object:{person}
```

4.2.3 Module type literals

Module references are specified by the interfaces they demand as a comma-delimited list.

```
module:{fungible-v2,user.votable}
```

4.3 Dereference operator

The dereference operator `::` allows a member of an interface specified in the type of a *module reference* to be invoked at run-time.

```
(interface baz
  (defun quux:bool (a:integer b:string))
  (defconst ONE 1)
  )
...
(defun foo (bar:module{baz})
  (bar::quux 1 "hi") ;; invokes 'quux' on whatever module is passed in
  bar::ONE           ;; directly references interface const
)
```

4.3.1 What can be typed

Function arguments and return types

```
(defun prefix:string (pfx:string str:string) (+ pfx str))
```

Let variables

```
(let ((a:integer 1) (b:integer 2)) (+ a b))
```

Tables and objects

Tables and objects can only take a schema type literal.

```
(deftable accounts:{account})

(defun get-order:{order} (id) (read orders id))
```

Consts

```
(defconst PENNY:decimal 0.1)
```

4.4 Special forms

4.4.1 Docs and Metadata

Many special forms like *defun* accept optional documentation strings, in the following form:

```
(defun average (a b)
  "take the average of a and b"
  (/ (+ a b) 2))
```

Alternately, users can specify metadata using a special `@`-prefix syntax. Supported metadata fields are `@doc` to provide a documentation string, and `@model` that can be used by Pact tooling to verify the correctness of the implementation:

```
(defun average (a b)
  @doc   "take the average of a and b"
  @model (property (= (+ a b) (* 2 result)))
  (/ (+ a b) 2))
```

Indeed, a bare docstring like `"foo"` is actually just a short form for `@doc "foo"`.

Specific information on *Properties* can be found in The Pact Property Checking System.

4.4.2 bless

```
(bless HASH)
```

Within a module declaration, bless a previous version of that module as identified by HASH. See *Dependency management* for a discussion of the blessing mechanism.

```
(module provider 'keyset
  (bless "ZHD9IZg-ro1wbx7dXi3Fr-CVmA-Pt71Ov9M1UNhzAkY")
  (bless "bctSHEz4N5Y1XQaic6eOoBmjty88HMMGfAdQLPuIGMw")
  ...
)
```

4.4.3 defun

```
(defun NAME ARGLIST [DOC-OR-META] BODY...)
```

Define NAME as a function, accepting ARGLIST arguments, with optional DOC-OR-META. Arguments are in scope for BODY, one or more expressions.

```
(defun add3 (a b c) (+ a (+ b c)))

(defun scale3 (a b c s)
  "multiply sum of A B C times s"
  (* s (add3 a b c)))
```

4.4.4 defcap

```
(defcap NAME ARGLIST [DOC] BODY...)
```

Define NAME as a capability, specified using ARGLIST arguments, with optional DOC. A `defcap` models a capability token which will be stored in the environment to represent some ability or right. Code in BODY is only called within special capability-related functions `with-capability` and `compose-capability` when the token as parameterized by the arguments supplied is not found in the environment. When executed, arguments are in scope for BODY, one or more expressions.

```
(defcap USER_GUARD (user)
  "Enforce user account guard
  (with-read accounts user
    { "guard": guard }
    (enforce-guard guard)))
```

4.4.5 defconst

```
(defconst NAME VALUE [DOC-OR-META])
```

Define NAME as VALUE, with option DOC-OR-META. Value is evaluated upon module load and "memoized".

```
(defconst COLOR_RED="#FF0000" "Red in hex")
(defconst COLOR_GRN="#00FF00" "Green in hex")
(defconst PI 3.14159265 "Pi to 8 decimals")
```

4.4.6 defpact

```
(defpact NAME ARGLIST [DOC-OR-META] STEPS...)
```

Define NAME as a *pact*, a computation comprised of multiple steps that occur in distinct transactions. Identical to *defun* except body must be comprised of *steps* to be executed in strict sequential order. Steps must uniformly be "public" (no entity indicator) or "private" (with entity indicator). With private steps, failures result in a reverse-sequence "rollback cascade".

```
(defpact payment (payer payer-entity payee
                  payee-entity amount)
  (step-with-rollback payer-entity
    (debit payer amount)
    (credit payer amount))
  (step payee-entity
    (credit payee amount)))
```

Public defpacts may be nested (though the recursion restrictions apply, so it must be a different defpact). They may be kicked off like a regular function call within a defpact, but are continued after the first step by calling `continue` with the same arguments.

As such, they have the following restrictions: - The number of steps of the child must match the number of steps of the parent. - If a parent defpact step has the rollback field, so must the child. If parent steps roll back, so do child steps. - `continue` must be called with the same continuation arguments as the defpact originally dispatched, to support multiple nested defpacts of the same function but with different arguments.

The following example shows well-formed defpacts with equal number of steps, nested rollbacks and continue:

```
(defpact payment (payer payee amount)
  (step-with-rollback
    (debit payer amount)
    (credit payer amount))
  (step payee-entity
    (credit payee amount)))

...
(defpact split-payment (payer payee1 payee2 amount ratio)
  (step-with-rollback
    (let
      ((payment1 (payment payer payee1 (* amount ratio)))
      (payment2 (payment payer payee2 (* amount (- 1 ratio))))
      )
      "step 0 complete"
    )
    (let
      ((payment1 (continue (payment payer payee1 (* amount ratio))))
       (payment2 (continue (payment payer payee2 (* amount (- 1 ratio)))))
      )
      "step 0 rolled back"
    )
  )
  (step
    (let
      ((payment1 (continue (payment payer payee1 (* amount ratio))))
       (payment2 (continue (payment payer payee2 (* amount (- 1 ratio)))))
      )
      "step 1 complete"
    )
  )
)
```

4.4.7 defschema

```
(defschema NAME [DOC-OR-META] FIELDS...)
```

Define NAME as a *schema*, which specifies a list of FIELDS. Each field is in the form `FIELDNAME[:FIELDTYPE]`.

```
(defschema accounts
  "Schema for accounts table".
  balance:decimal
  amount:decimal
  ccy:string
  data)
```

4.4.8 deftable

```
(deftable NAME[:SCHEMA] [DOC-OR-META])
```

Define NAME as a *table*, used in database functions. Note the table must still be created with create-table.

4.4.9 let

```
(let (BINDPAIR [BINDPAIR [...]]) BODY)
```

Bind variables in BINDPAIRs to be in scope over BODY. Variables within BINDPAIRs cannot refer to previously-declared variables in the same let binding; for this use *let**.

```
(let ((x 2)
      (y 5))
  (* x y))
> 10
```

4.4.10 let*

```
(let* (BINDPAIR [BINDPAIR [...]]) BODY)
```

Bind variables in BINDPAIRs to be in scope over BODY. Variables can reference previously declared BINDPAIRS in the same let. `let*` is expanded at compile-time to nested `let` calls for each BINDPAIR; thus `let` is preferred where possible.

```
(let* ((x 2)
       (y (* x 10)))
  (+ x y))
> 22
```

4.4.11 cond;

```
(cond (TEST BRANCH) [(TEST2 BRANCH2) [...]] ELSE-BRANCH)
```

Special form/sugar to produce a series of "if-elseif-else" expressions, such that if TEST1 passes, BRANCH1 is evaluated, otherwise followed by evaluating TEST2 -> BRANCH2 etc. ELSE-BRANCH is evaluated if all tests fail.

`cond` is syntactically expanded such that

```
(cond
   (a b)
   (c d)
   (e f)
   g)
```

is expanded to:

```
(if a b (if c d (if e f g)))
```

4.4.12 step

```
(step EXPR)
(step ENTITY EXPR)
```

Define a step within a *defpact*, such that any prior steps will be executed in prior transactions, and later steps in later transactions. Including an ENTITY argument indicates that this step is intended for confidential transactions. Therefore, only the ENTITY would execute the step, and other participants would "skip" it.

4.4.13 step-with-rollback

```
(step-with-rollback EXPR ROLLBACK-EXPR)
(step-with-rollback ENTITY EXPR ROLLBACK-EXPR)
```

Define a step within a *defpact* similarly to *step* but specifying ROLLBACK-EXPR. With ENTITY, ROLLBACK-EXPR will only be executed upon failure of a subsequent step, as part of a reverse-sequence "rollback cascade" going back from the step that failed to the first step. Without ENTITY, ROLLBACK-EXPR functions as a "cancel function" to be explicitly executed by a participant.

4.4.14 use

```
(use MODULE)
(use MODULE HASH)
(use MODULE IMPORTS)
(use MODULE HASH IMPORTS)
```

Import an existing MODULE into a namespace. Can only be issued at the top-level, or within a module declaration. MODULE can be a string, symbol or bare atom. With HASH, validate that the imported module's hash matches HASH, failing if not. Use describe-module to query for the hash of a loaded module on the chain.

An optional list of IMPORTS consisting of function, constant, and schema names may be supplied. When this explicit import list is present, only those names will be made available for use in the module body. If no list is supplied, then every name in the imported module will be brought into scope. When two modules are defined in the same transaction, all names will be in scope for all modules, and import behavior will be defaulted to the entire module. IMPORTS may only be empty when a module hash is also supplied. If a module hash is not supplied, IMPORTS are required to be either a non-empty list, or left undeclared.

```
(use accounts)
(transfer "123" "456" 5 (time "2016-07-22T11:26:35Z"))
"Write succeeded"
```

```
(use accounts "ToV3sYFMghd7AN1TFKdWk_w00HjUepVlqKL79ckHG_s")
(transfer "123" "456" 5 (time "2016-07-22T11:26:35Z"))
"Write succeeded"
```

```
(use accounts [ transfer example-fun ])
(transfer "123" "456" 5 (time "2016-07-22T11:26:35Z"))
"Write succeeded"
```

```
(use accounts "ToV3sYFMghd7AN1TFKdWk_w00HjUepVlqKL79ckHG_s" [ transfer example-fun ])
(transfer "123" "456" 5 (time "2016-07-22T11:26:35Z"))
"Write succeeded"
```

4.4.15 interface

```
(interface NAME [DOR-OR-META] BODY...)
```

Define and install interface NAME, with optional DOC-OR-META.

BODY is composed of definitions that will be scoped in the module. Valid expressions in a module include:

- *defun*

- *defconst*
- *defschema*
- *defpact*
- *defcap*
- *use*
- *models*

```
(interface coin-sig
  "'coin-sig' represents the Kadena Coin Contract interface. This contract     \
  \provides both the the general interface for a Kadena's token, supplying a   \
  \transfer function, coinbase, account creation and balance query."
  (defun create-account:string (account:string guard:guard)
    @doc "Create an account for ACCOUNT, with GUARD controlling access to the  \
    \account."
    @model [ (property (not (= account ""))) ]
    )
  (defun transfer:string (sender:string receiver:string amount:decimal)
    @doc "Transfer AMOUNT between accounts SENDER and RECEIVER on the same    \
    \chain. This fails if either SENDER or RECEIVER does not exist.           \
    \Create-on-transfer can be done using the 'transfer-and-create' function."
    @model [ (property (> amount 0.0))
             (property (not (= sender receiver)))
           ]
    )
  (defun account-balance:decimal (account:string)
    @doc "Check an account's balance"
    @model [ (property (not (= account ""))) ]
    )
)
```

4.4.16 module

```
(module NAME KEYSET-OR-GOVERNANCE [DOC-OR-META] BODY...)
```

Define and install module NAME, with module admin governed by KEYSET-OR-GOVERNANCE, with optional DOC-OR-META.

If KEYSET-OR-GOVERNANCE is a string, it references a keyset that has been installed with `define-keyset` that will be tested whenever module admin is required. If KEYSET-OR-GOVERNANCE is an unqualified atom, it references a `defcap` capability which will be acquired if module admin is requested.

BODY is composed of definitions that will be scoped in the module. Valid productions in a module include:

- *defun*
- *defpact*
- *defcap*
- *deftable*
- *defschema*
- *defconst*
- *implements*

- *use*
- *bless*

```
(module accounts 'accounts-admin
  "Module for interacting with accounts"

  (defun create-account (id bal)
   "Create account ID with initial balance BAL"
   (insert accounts id { "balance": bal }))

  (defun transfer (from to amount)
   "Transfer AMOUNT from FROM to TO"
   (with-read accounts from { "balance": fbal }
    (enforce (<= amount fbal) "Insufficient funds")
     (with-read accounts to { "balance": tbal }
      (update accounts from { "balance": (- fbal amount) })
      (update accounts to { "balance": (+ tbal amount) }))))
)
```

4.4.17 implements

```
(implements INTERFACE)
```

Specify that containing module *implements* interface INTERFACE. This requires the module to implement all functions, pacts, and capabilities specified in INTERFACE with identical signatures (same argument names and declared types).

Note that models declared for the implemented interface and its members will be appended to whatever models are declared within the implementing module.

A module thus specified can be used as a *module reference* for the specified interface(s).

4.5 Expressions

Expressions may be *literals*, atoms, s-expressions, or references.

4.5.1 Atoms

Atoms are non-reserved barewords starting with a letter or allowed symbol, and containing letters, digits and allowed symbols. Allowed symbols are `%#+-_&$@<>=?*!|/`. Atoms must resolve to a variable bound by a *defun*, *defpact*, *binding* form, *lambda* form, or to symbols imported into the namespace with *use*.

4.5.2 S-expressions

S-expressions are formed with parentheses, with the first atom determining if the expression is a *special form* or a function application, in which case the first atom must refer to a definition.

Partial application

An application with less than the required arguments is in some contexts a valid *partial application* of the function. However, this is only supported in Pact's *functional-style functions*; anywhere else this will result in a runtime error.

4.5.3 References

References are multiple atoms joined by a dot `.` that directly resolve to definitions found in other modules.

```
pact> accounts.transfer
"(defun accounts.transfer (src,dest,amount,date) \"transfer AMOUNT from
SRC to DEST\")"
pact> transfer
Eval failure:
transfer<EOF>: Cannot resolve transfer
pact> (use 'accounts)
"Using \"accounts\""
pact> transfer
"(defun accounts.transfer (src,dest,amount,date) \"transfer AMOUNT from
SRC to DEST\")"
```

References are preferred over `use` for transactions, as references resolve faster. However, when defining a module, `use` is preferred for legibility.

CHAPTER 5

Time formats

The parse-time and format-time functions accept format codes that derive from GNU `strftime` with some extensions, as follows:

`%%` - literal `"%"`

`%z` - RFC 822/ISO 8601:1988 style numeric time zone (e.g., `"-0600"` or `"+0100"`)

`%N` - ISO 8601 style numeric time zone (e.g., `"-06:00"` or `"+01:00"`) /EXTENSION/

`%Z` - timezone name

`%c` - The preferred calendar time representation for the current locale. As 'dateTimeFmt' `locale` (e.g. `%a %b %e %H:%M:%S %Z %Y`)

`%R` - same as `%H:%M`

`%T` - same as `%H:%M:%S`

`%X` - The preferred time of day representation for the current locale. As 'timeFmt' `locale` (e.g. `%H:%M:%S`)

`%r` - The complete calendar time using the AM/PM format of the current locale. As 'time12Fmt' `locale` (e.g. `%I:%M:%S %p`)

`%P` - day-half of day from ('amPm' `locale`), converted to lowercase, `"am"`, `"pm"`

`%p` - day-half of day from ('amPm' `locale`), `"AM"`, `"PM"`

`%H` - hour of day (24-hour), 0-padded to two chars, `"00"-"23"`

`%k` - hour of day (24-hour), space-padded to two chars, `" 0"-"23"`

`%I` - hour of day-half (12-hour), 0-padded to two chars, `"01"-"12"`

`%l` - hour of day-half (12-hour), space-padded to two chars, `" 1"-"12"`

`%M` - minute of hour, 0-padded to two chars, `"00"-"59"`

`%S` - second of minute (without decimal part), 0-padded to two chars, `"00"-"60"`

`%v` - microsecond of second, 0-padded to six chars, `"000000"-"999999"`. /EXTENSION/

`%Q` - decimal point and fraction of second, up to 6 second decimals, without trailing zeros. For a whole number of seconds, `%Q` produces the empty string. /EXTENSION/

`%s` - number of whole seconds since the Unix epoch. For times before the Unix epoch, this is a negative number. Note that in `%s.%q` and `%s%Q` the decimals are positive, not negative. For example, 0.9 seconds before the Unix epoch is formatted as `"-1.1"` with `%s%Q`.

`%D` - same as `%m\/%d\/%y`

`%F` - same as `%Y-%m-%d`

`%x` - as 'dateFmt' `locale` (e.g. `%m\/%d\/%y`)

`%Y` - year, no padding.

`%y` - year of century, 0-padded to two chars, `"00"-"99"`

`%C` - century, no padding.

`%B` - month name, long form ('fst' from 'months' `locale`), `"January"-"December"`

`%b`, `%h` - month name, short form ('snd' from 'months' `locale`), `"Jan"-"Dec"`

`%m` - month of year, 0-padded to two chars, `"01"-"12"`

`%d` - day of month, 0-padded to two chars, `"01"-"31"`

`%e` - day of month, space-padded to two chars, `" 1"-"31"`

`%j` - day of year, 0-padded to three chars, `"001"-"366"`

`%G` - year for Week Date format, no padding.

`%g` - year of century for Week Date format, 0-padded to two chars, `"00"-"99"`

`%f` - century for Week Date format, no padding. /EXTENSION/

`%V` - week of year for Week Date format, 0-padded to two chars, `"01"-"53"`

`%u` - day of week for Week Date format, `"1"-"7"`

`%a` - day of week, short form ('snd' from 'wDays' `locale`), `"Sun"-"Sat"`

`%A` - day of week, long form ('fst' from 'wDays' `locale`), `"Sunday"-"Saturday"`

`%U` - week of year where weeks start on Sunday (as 'sundayStartWeek'), 0-padded to two chars, `"00"-"53"`

`%w` - day of week number, `"0"` (= Sunday) – `"6"` (= Saturday)

`%W` - week of year where weeks start on Monday (as 'Data.Thyme.Calendar.WeekdayOfMonth.mondayStartWeek'), 0-padded to two chars, `"00"-"53"`

Note: `%q` (picoseconds, zero-padded) does not work properly so not documented here.

5.1 Default format and JSON serialization

The default format is a UTC ISO8601 date+time format: "%Y-%m-%dT%H:%M:%SZ", as accepted by the time function. While the time object internally supports up to microsecond resolution, values returned from the Pact interpreter as JSON will be serialized with the default format. When higher resolution is desired, explicitly format times with `%v` and related codes.

5.2 Examples

5.2.1 ISO8601

```
pact> (format-time "%Y-%m-%dT%H:%M:%S%N" (time "2016-07-23T13:30:45Z"))
"2016-07-23T13:30:45+00:00"
```

5.2.2 RFC822

```
pact> (format-time "%a, %_d %b %Y %H:%M:%S %Z" (time "2016-07-23T13:30:45Z"))
"Sat, 23 Jul 2016 13:30:45 UTC"
```

5.2.3 YYYY-MM-DD hh:mm:ss.000000

```
pact> (format-time "%Y-%m-%d %H:%M:%S.%v" (add-time (time "2016-07-23T13:30:45Z") 0.
↪001002))
"2016-07-23 13:30:45.001002"
```

CHAPTER 6

Built-in Functions

6.1 General

6.1.1 CHARSET_ASCII

Constant denoting the ASCII charset

Constant: `CHARSET_ASCII:integer = 0`

6.1.2 CHARSET_LATIN1

Constant denoting the Latin-1 charset ISO-8859-1

Constant: `CHARSET_LATIN1:integer = 1`

6.1.3 at

idx `integer` *list* `[<l>]` → `<a>`

idx `string` *object* `object:<{o}>` → `<a>`

Index LIST at IDX, or get value with key IDX from OBJECT.

```
pact> (at 1 [1 2 3])
2
pact> (at "bar" { "foo": 1, "bar": 2 })
2
```

6.1.4 base64-decode

string `string` → `string`

Decode STRING from unpadded base64

```
pact> (base64-decode "aGVsbG8gd29ybGQh")
"hello world!"
```

6.1.5 base64-encode

string `string` → `string`

Encode STRING as unpadded base64

```
pact> (base64-encode "hello world!")
"aGVsbG8gd29ybGQh"
```

6.1.6 bind

src `object:<{row}>` *binding* `binding:<{row}>` → <a>

Special form evaluates SRC to an object which is bound to with BINDINGS over subsequent body statements.

```
pact> (bind { "a": 1, "b": 2 } { "a" := a-value } a-value)
1
```

6.1.7 chain-data

→ `object:{public-chain-data}`

Get transaction public metadata. Returns an object with 'chain-id', 'block-height', 'block-time', 'prev-block-hash', 'sender', 'gas-limit', 'gas-price', and 'gas-fee' fields.

```
pact> (chain-data)
{"block-height": 0,"block-time": "1970-01-01T00:00:00Z","chain-id": "","gas-limit": 0,
↪"gas-price": 0.0,"prev-block-hash": "","sender": ""}
```

6.1.8 compose

x `x:<a>` `->` <b> *y* `x:<b>` `->` <c> *value* <a> → <c>

Compose X and Y, such that X operates on VALUE, and Y on the results of X.

```
pact> (filter (compose (length) (< 2)) ["my" "dog" "has" "fleas"])
["dog" "has" "fleas"]
```

6.1.9 concat

str-list `[string]` → `string`

Takes STR-LIST and concats each of the strings in the list, returning the resulting string

```
pact> (concat ["k" "d" "a"])
"kda"
pact> (concat (map (+ " ") (str-to-list "abcde")))
" a b c d e"
```

6.1.10 constantly

value <a> *ignore1* <b> → <a>

value <a> *ignore1* <b> *ignore2* <c> → <a>

value <a> *ignore1* <b> *ignore2* <c> *ignore3* <d> → <a>

Lazily ignore arguments IGNORE* and return VALUE.

```
pact> (filter (constantly true) [1 2 3])
[1 2 3]
```

6.1.11 contains

value <a> *list* `[<a>]` → `bool`

key <a> *object* `object:<{o}>` → `bool`

value `string` *string* `string` → `bool`

Test that LIST or STRING contains VALUE, or that OBJECT has KEY entry.

```
pact> (contains 2 [1 2 3])
true
pact> (contains 'name { 'name: "Ted", 'age: 72 })
true
pact> (contains "foo" "foobar")
true
```

6.1.12 continue

value *a* → *a*

Continue a previously started nested defpact.

```
(continue (coin.transfer-crosschain "bob" "alice" 10.0))
```

6.1.13 define-namespace

namespace `string` *user-guard* `guard` *admin-guard* `guard` → `string`

Create a namespace called NAMESPACE where ownership and use of the namespace is controlled by GUARD. If NAMESPACE is already defined, then the guard previously defined in NAMESPACE will be enforced, and GUARD will be rotated in its place.

```
(define-namespace 'my-namespace (read-keyset 'user-ks) (read-keyset 'admin-ks))
```

Top level only: this function will fail if used in module code.

6.1.14 distinct

values `[<a>]` → `[<a>]`

Returns from a homogeneous list of VALUES a list with duplicates removed. The original order of the values is preserved.

```
pact> (distinct [3 3 1 1 2 2])
[3 1 2]
```

6.1.15 drop

count `integer` *list* `<a[[<l>],string]>` → `<a[[<l>],string]>`

keys `[string]` *object* `object:<{o}>` → `object:<{o}>`

Drop COUNT values from LIST (or string), or entries having keys in KEYS from OBJECT. If COUNT is negative, drop from end. If COUNT exceeds the interval (-2^{63},2^{63}), it is truncated to that range.

```
pact> (drop 2 "vwxyz")
"xyz"
pact> (drop (- 2) [1 2 3 4 5])
[1 2 3]
pact> (drop ['name] { 'name: "Vlad", 'active: false})
{"active": false}
```

6.1.16 enforce

test `bool` *msg* `string` → `bool`

Fail transaction with MSG if pure expression TEST is false. Otherwise, returns true.

```
pact> (enforce (!= (+ 2 2) 4) "Chaos reigns")
<interactive>:0:0: Chaos reigns
```

6.1.17 enforce-one

msg `string` *tests* `[bool]` → `bool`

Run TESTS in order (in pure context, plus keyset enforces). If all fail, fail transaction. Short-circuits on first success.

```
pact> (enforce-one "Should succeed on second test" [(enforce false "Skip me")␣
↪(enforce (= (+ 2 2) 4) "Chaos reigns")])
true
```

6.1.18 enforce-pact-version

min-version `string` → `bool`

min-version `string` *max-version* `string` → `bool`

Enforce runtime pact version as greater than or equal MIN-VERSION, and less than or equal MAX-VERSION. Version values are matched numerically from the left, such that '2', '2.2', and '2.2.3' would all allow '2.2.3'.

```
pact> (enforce-pact-version "2.3")
true
```

Top level only: this function will fail if used in module code.

6.1.19 enumerate

from `integer` *to* `integer` *inc* `integer` → `[integer]`

from `integer` *to* `integer` → `[integer]`

Returns a sequence of numbers from FROM to TO (both inclusive) as a list. INC is the increment between numbers in the sequence. If INC is not given, it is assumed to be 1. Additionally, if INC is not given and FROM is greater than TO assume a value for INC of -1. If FROM equals TO, return the singleton list containing FROM, irrespective of INC's value. If INC is equal to zero, this function will return the singleton list containing FROM. If INC is such that FROM + INC > TO (when FROM < TO) or FROM + INC < TO (when FROM > TO) return the singleton list containing FROM. Lastly, if INC is such that FROM + INC < TO (when FROM < TO) or FROM + INC > TO (when FROM > TO), then this function fails.

```
pact> (enumerate 0 10 2)
[0 2 4 6 8 10]
pact> (enumerate 0 10)
[0 1 2 3 4 5 6 7 8 9 10]
pact> (enumerate 10 0)
[10 9 8 7 6 5 4 3 2 1 0]
```

6.1.20 filter

app `x:<a> -> bool` *list* `[<a>]` → `[<a>]`

Filter LIST by applying APP to each element. For each true result, the original value is kept.

```
pact> (filter (compose (length) (< 2)) ["my" "dog" "has" "fleas"])
["dog" "has" "fleas"]
```

6.1.21 fold

app `x:<a> y:<b> -> <a>` *init* `<a>` *list* `[<b>]` → `<a>`

Iteratively reduce LIST by applying APP to last result and element, starting with INIT.

```
pact> (fold (+) 0 [100 10 5])
115
```

6.1.22 format

template `string` *vars* `[*]` → `string`

Interpolate VARS into TEMPLATE using {}.

```
pact> (format "My {} has {}" ["dog" "fleas"])
"My dog has fleas"
```

6.1.23 hash

value <a> → `string`

Compute BLAKE2b 256-bit hash of VALUE represented in unpadded base64-url. Strings are converted directly while other values are converted using their JSON representation. Non-value-level arguments are not allowed.

```
pact> (hash "hello")
"Mk3PAn3UowqTLEQfNlol6GsXPe-kuOWJSCU0cbgbcs8"
pact> (hash { 'foo: 1 })
"h9BZgylRf_M4HxcBXr15IcSXXXSz74ZC2IAViGle_z4"
```

6.1.24 identity

value <a> → <a>

Return provided value.

```
pact> (map (identity) [1 2 3])
[1 2 3]
```

6.1.25 if

cond `bool` *then* <a> *else* <a> → <a>

Test COND. If true, evaluate THEN. Otherwise, evaluate ELSE.

```
pact> (if (= (+ 2 2) 4) "Sanity prevails" "Chaos reigns")
"Sanity prevails"
```

6.1.26 int-to-str

base `integer` *val* `integer` → `string`

Represent integer VAL as a string in BASE. BASE can be 2-16, or 64 for unpadded base64URL. Only positive values are allowed for base64URL conversion.

```
pact> (int-to-str 16 65535)
"ffff"
pact> (int-to-str 64 43981)
"q80"
```

6.1.27 is-charset

charset `integer` *input* `string` → `bool`

Check that a string INPUT conforms to the a supported character set CHARSET. Character sets currently supported are: 'CHARSET_LATIN1' (ISO-8859-1), and 'CHARSET_ASCII' (ASCII). Support for sets up through ISO 8859-5 supplement will be added in the future.

```
pact> (is-charset CHARSET_ASCII "hello world")
true
pact> (is-charset CHARSET_ASCII "I am nÖt ascii")
false
pact> (is-charset CHARSET_LATIN1 "I am nÖt ascii, but I am latin1!")
true
```

6.1.28 length

x `<a[[<l>],string,object:<{o}>]>` → `integer`

Compute length of X, which can be a list, a string, or an object.

```
pact> (length [1 2 3])
3
pact> (length "abcdefgh")
8
pact> (length { "a": 1, "b": 2 })
2
```

6.1.29 list

elems `*` → `[*]`

Create list from ELEMS. Deprecated in Pact 2.1.1 with literal list support.

```
pact> (list 1 2 3)
[1 2 3]
```

6.1.30 list-modules

→ `[string]`

List modules available for loading.

Top level only: this function will fail if used in module code.

6.1.31 make-list

length `integer` *value* `<a>` → `[<a>]`

Create list by repeating VALUE LENGTH times.

```
pact> (make-list 5 true)
[true true true true true]
```

6.1.32 map

app `x:<b> -> <a>` *list* `[<b>]` → `[<a>]`

Apply APP to each element in LIST, returning a new list of results.

```
pact> (map (+ 1) [1 2 3])
[2 3 4]
```

6.1.33 namespace

namespace `string` → `string`

Set the current namespace to NAMESPACE. All expressions that occur in a current transaction will be contained in NAMESPACE, and once committed, may be accessed via their fully qualified name, which will include the namespace. Subsequent namespace calls in the same tx will set a new namespace for all declarations until either the next namespace declaration, or the end of the tx.

```
(namespace 'my-namespace)
```

Top level only: this function will fail if used in module code.

6.1.34 pact-id

→ `string`

Return ID if called during current pact execution, failing if not.

6.1.35 pact-version

→ `string`

Obtain current pact build version.

```
pact> (pact-version)
"4.3"
```

Top level only: this function will fail if used in module code.

6.1.36 public-chain-data

Schema type for data returned from 'chain-data'.

Fields: `chain-id:string` `block-height:integer` `block-time:time` `prev-block-hash:string` `sender:string` `gas-limit:integer` `gas-price:decimal`

6.1.37 read-decimal

key `string` → `decimal`

Parse KEY string or number value from top level of message data body as decimal.

```
(defun exec ()
   (transfer (read-msg "from") (read-msg "to") (read-decimal "amount")))
```

6.1.38 read-integer

key `string` → `integer`

Parse KEY string or number value from top level of message data body as integer.

```
(read-integer "age")
```

6.1.39 read-msg

→ <a>

key `string` → <a>

Read KEY from top level of message data body, or data body itself if not provided. Coerces value to their corresponding pact type: String -> string, Number -> integer, Boolean -> bool, List -> list, Object -> object.

```
(defun exec ()
   (transfer (read-msg "from") (read-msg "to") (read-decimal "amount")))
```

6.1.40 read-string

key `string` → `string`

Parse KEY string or number value from top level of message data body as string.

```
(read-string "sender")
```

6.1.41 remove

key `string` *object* `object:<{o}>` → `object:<{o}>`

Remove entry for KEY from OBJECT.

```
pact> (remove "bar" { "foo": 1, "bar": 2 })
{"foo": 1}
```

6.1.42 resume

binding `binding:<{r}>` → <a>

Special form binds to a yielded object value from the prior step execution in a pact. If yield step was executed on a foreign chain, enforce endorsement via SPV.

6.1.43 reverse

list [<a>] → [<a>]

Reverse LIST.

```
pact> (reverse [1 2 3])
[3 2 1]
```

6.1.44 sort

values `[<a>]` → `[<a>]`

fields `[string]` *values* `[object:<{o}>]` → `[object:<{o}>]`

Sort a homogeneous list of primitive VALUES, or objects using supplied FIELDS list.

```
pact> (sort [3 1 2])
[1 2 3]
pact> (sort ['age] [{'name: "Lin",'age: 30} {'name: "Val",'age: 25}])
[{"name": "Val","age": 25} {"name": "Lin","age": 30}]
```

6.1.45 str-to-int

str-val `string` → `integer`

base `integer` *str-val* `string` → `integer`

Compute the integer value of STR-VAL in base 10, or in BASE if specified. STR-VAL can be up to 512 chars in length. BASE must be between 2 and 16, or 64 to perform unpadded base64url conversion. Each digit must be in the correct range for the base.

```
pact> (str-to-int 16 "abcdef123456")
188900967593046
pact> (str-to-int "123456")
123456
pact> (str-to-int 64 "q80")
43981
```

6.1.46 str-to-list

str `string` → `[string]`

Takes STR and returns a list of single character strings

```
pact> (str-to-list "hello")
["h" "e" "l" "l" "o"]
pact> (concat (map (+ " ") (str-to-list "abcde")))
" a b c d e"
```

6.1.47 take

count `integer` *list* `<a[[<l>],string]>` → `<a[[<l>],string]>`

keys `[string]` *object* `object:<{o}>` → `object:<{o}>`

Take COUNT values from LIST (or string), or entries having keys in KEYS from OBJECT. If COUNT is negative, take from end. If COUNT exceeds the interval (-2^{63},^{2}63), it is truncated to that range.

```
pact> (take 2 "abcd")
"ab"
pact> (take (- 3) [1 2 3 4 5])
[3 4 5]
pact> (take ['name] { 'name: "Vlad", 'active: false})
{"name": "Vlad"}
```

6.1.48 try

default <a> *action* <a> → <a>

Attempt a pure ACTION, returning DEFAULT in the case of failure. Pure expressions are expressions which do not do i/o or work with non-deterministic state in contrast to impure expressions such as reading and writing to a table.

```
pact> (try 3 (enforce (= 1 2) "this will definitely fail"))
3
(expect "impure expression fails and returns default" "default" (try "default" (with-
↪read accounts id {'ccy := ccy}) ccy))
```

6.1.49 tx-hash

→ string

Obtain hash of current transaction as a string.

```
pact> (tx-hash)
"DldRwCblQ7Loqy6wYJnaodHl30d3j3eH-qtFzfEv46g"
```

6.1.50 typeof

x <a> → string

Returns type of X as string.

```
pact> (typeof "hello")
"string"
```

6.1.51 where

field string *app* x:<a> -> bool *value* object:<{row}> → bool

Utility for use in 'filter' and 'select' applying APP to FIELD in VALUE.

```
pact> (filter (where 'age (> 20)) [{'name: "Mary",'age: 30} {'name: "Juan",'age: 15}])
[{"name": "Juan","age": 15}]
```

6.1.52 yield

object object:<{y}> → object:<{y}>

object object:<{y}> *target-chain* string → object:<{y}>

Yield OBJECT for use with 'resume' in following pact step. With optional argument TARGET-CHAIN, target subsequent step to execute on targeted chain using automated SPV endorsement-based dispatch.

```
(yield { "amount": 100.0 })
(yield { "amount": 100.0 } "some-chain-id")
```

6.1.53 zip

f `x:<a> y:<b> -> <c>` *list1* `[<a>]` *list2* `[<b>]` → `[<c>]`

Combine two lists with some function f, into a new list, the length of which is the length of the shortest list.

```
pact> (zip (+) [1 2 3 4] [4 5 6 7])
[5 7 9 11]
pact> (zip (-) [1 2 3 4] [4 5 6])
[-3 -3 -3]
pact> (zip (+) [1 2 3] [4 5 6 7])
[5 7 9]
pact> (zip (lambda (x y) { 'x: x, 'y: y }) [1 2 3 4] [4 5 6 7])
[{"x": 1,"y": 4} {"x": 2,"y": 5} {"x": 3,"y": 6} {"x": 4,"y": 7}]
```

6.2 Database

6.2.1 create-table

table `table:<{row}>` → `string`

Create table TABLE.

```
(create-table accounts)
```

Top level only: this function will fail if used in module code.

6.2.2 describe-keyset

keyset `string` → `object:*`

Get metadata for KEYSET.

Top level only: this function will fail if used in module code.

6.2.3 describe-module

module `string` → `object:*`

Get metadata for MODULE. Returns an object with 'name', 'hash', 'blessed', 'code', and 'keyset' fields.

```
(describe-module 'my-module)
```

Top level only: this function will fail if used in module code.

6.2.4 describe-table

table `table:<{row}>` → `object:*`

Get metadata for TABLE. Returns an object with 'name', 'hash', 'blessed', 'code', and 'keyset' fields.

```
(describe-table accounts)
```

Top level only: this function will fail if used in module code.

6.2.5 fold-db

table `table:<{row}>` *qry* `a:string b:object:<{row}> -> bool` *consumer* `a:string b:object:<{row}> -> <b>` → `[<b>]`

Select rows from TABLE using QRY as a predicate with both key and value, and then accumulate results of the query in CONSUMER. Output is sorted by the ordering of keys.

```
(let*
 ((qry (lambda (k obj) true)) ;; select all rows
  (f (lambda (x) [(at 'firstName x), (at 'b x)]))
 )
 (fold-db people (qry) (f))
)
```

6.2.6 insert

table `table:<{row}>` *key* `string` *object* `object:<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data, failing if data already exists for KEY.

```
(insert accounts id { "balance": 0.0, "note": "Created account." })
```

6.2.7 keylog

table `table:<{row}>` *key* `string` *txid* `integer` → `[object:*]`

Return updates to TABLE for a KEY in transactions at or after TXID, in a list of objects indexed by txid.

```
(keylog accounts "Alice" 123485945)
```

6.2.8 keys

table `table:<{row}>` → `[string]`

Return all keys in TABLE.

```
(keys accounts)
```

6.2.9 read

table `table:<{row}>` *key* `string` → `object:<{row}>`

table `table:<{row}>` *key* `string` *columns* `[string]` → `object:<{row}>`

Read row from TABLE for KEY, returning database record object, or just COLUMNS if specified.

```
(read accounts id ['balance 'ccy])
```

6.2.10 select

table `table:<{row}>` *where* `row:object:<{row}> -> bool` → `[object:<{row}>]`

table `table:<{row}>` *columns* `[string]` *where* `row:object:<{row}> -> bool` → `[object:<{row}>]`

Select full rows or COLUMNS from table by applying WHERE to each row to get a boolean determining inclusion.

```
(select people ['firstName,'lastName] (where 'name (= "Fatima")))
(select people (where 'age (> 30)))?
```

6.2.11 txids

table `table:<{row}>` *txid* `integer` → `[integer]`

Return all txid values greater than or equal to TXID in TABLE.

```
(txids accounts 123849535)
```

6.2.12 txlog

table `table:<{row}>` *txid* `integer` → `[object:*]`

Return all updates to TABLE performed in transaction TXID.

```
(txlog accounts 123485945)
```

6.2.13 update

table `table:<{row}>` *key* `string` *object* `object:~<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data, failing if data does not exist for KEY.

```
(update accounts id { "balance": (+ bal amount), "change": amount, "note": "credit" })
```

6.2.14 with-default-read

table `table:<{row}>` *key* `string` *defaults* `object:~<{row}>` *bindings* `binding:~<{row}>` → `<a>`

Special form to read row from TABLE for KEY and bind columns per BINDINGS over subsequent body statements. If row not found, read columns from DEFAULTS, an object with matching key names.

```
(with-default-read accounts id { "balance": 0, "ccy": "USD" } { "balance":= bal, "ccy
↪":= ccy }
  (format "Balance for {} is {} {}" [id bal ccy]))
```

6.2.15 with-read

table `table:<{row}>` *key* `string` *bindings* `binding:<{row}>` → `<a>`

Special form to read row from TABLE for KEY and bind columns per BINDINGS over subsequent body statements.

```
(with-read accounts id { "balance":= bal, "ccy":= ccy }
  (format "Balance for {} is {} {}" [id bal ccy]))
```

6.2.16 write

table `table:<{row}>` *key* `string` *object* `object:<{row}>` → `string`

Write entry in TABLE for KEY of OBJECT column data.

```
(write accounts id { "balance": 100.0 })
```

6.3 Time

6.3.1 add-time

time `time` *seconds* `decimal` → `time`

time `time` *seconds* `integer` → `time`

Add SECONDS to TIME; SECONDS can be integer or decimal.

```
pact> (add-time (time "2016-07-22T12:00:00Z") 15)
"2016-07-22T12:00:15Z"
```

6.3.2 days

n `decimal` → `decimal`

n `integer` → `decimal`

N days, for use with 'add-time'

```
pact> (add-time (time "2016-07-22T12:00:00Z") (days 1))
"2016-07-23T12:00:00Z"
```

6.3.3 diff-time

time1 `time` *time2* `time` → `decimal`

Compute difference between TIME1 and TIME2 in seconds.

```
pact> (diff-time (parse-time "%T" "16:00:00") (parse-time "%T" "09:30:00"))
23400.0
```

6.3.4 format-time

format `string` *time* `time` → `string`

Format TIME using FORMAT. See "Time Formats" docs for supported formats.

```
pact> (format-time "%F" (time "2016-07-22T12:00:00Z"))
"2016-07-22"
```

6.3.5 hours

n `decimal` → `decimal`

n `integer` → `decimal`

N hours, for use with 'add-time'

```
pact> (add-time (time "2016-07-22T12:00:00Z") (hours 1))
"2016-07-22T13:00:00Z"
```

6.3.6 minutes

n `decimal` → `decimal`

n `integer` → `decimal`

N minutes, for use with 'add-time'.

```
pact> (add-time (time "2016-07-22T12:00:00Z") (minutes 1))
"2016-07-22T12:01:00Z"
```

6.3.7 parse-time

format `string` *utcval* `string` → `time`

Construct time from UTCVAL using FORMAT. See "Time Formats" docs for supported formats.

```
pact> (parse-time "%F" "2016-09-12")
"2016-09-12T00:00:00Z"
```

6.3.8 time

utcval `string` → `time`

Construct time from UTCVAL using ISO8601 format (%Y-%m-%dT%H:%M:%SZ).

```
pact> (time "2016-07-22T11:26:35Z")
"2016-07-22T11:26:35Z"
```

6.4 Operators

6.4.1 !=

x `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset,guard,module{}]>` *y* `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset,guard,module{}]>` → `bool`

True if X does not equal Y.

```
pact> (!= "hello" "goodbye")
true
```

6.4.2 & {#&}

x `integer` *y* `integer` → `integer`

Compute bitwise X and Y.

```
pact> (& 2 3)
2
pact> (& 5 -7)
1
```

6.4.3 *

x `<a[integer,decimal]>` *y* `<a[integer,decimal]>` → `<a[integer,decimal]>`

x `<a[integer,decimal]>` *y* `<b[integer,decimal]>` → `decimal`

Multiply X by Y.

```
pact> (* 0.5 10.0)
5.0
pact> (* 3 5)
15
```

6.4.4 +

x `<a[integer,decimal]>` *y* `<a[integer,decimal]>` → `<a[integer,decimal]>`

x `<a[integer,decimal]>` *y* `<b[integer,decimal]>` → `decimal`

x `<a[string,[<l>],object:<{o}>]>` *y* `<a[string,[<l>],object:<{o}>]>` → `<a[string,[<l>],object:<{o}>]>`

Add numbers, concatenate strings/lists, or merge objects.

```
pact> (+ 1 2)
3
pact> (+ 5.0 0.5)
5.5
pact> (+ "every" "body")
"everybody"
pact> (+ [1 2] [3 4])
[1 2 3 4]
pact> (+ { "foo": 100 } { "foo": 1, "bar": 2 })
{"bar": 2,"foo": 100}
```

6.4.5 -

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

x <a[integer,decimal]> → <a[integer,decimal]>

Negate X, or subtract Y from X.

```
pact> (- 1.0)
-1.0
pact> (- 3 2)
1
```

6.4.6 /

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

Divide X by Y.

```
pact> (/ 10.0 2.0)
5.0
pact> (/ 8 3)
2
```

6.4.7 <

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X < Y.

```
pact> (< 1 3)
true
pact> (< 5.24 2.52)
false
pact> (< "abc" "def")
true
```

6.4.8 <=

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X <= Y.

```
pact> (<= 1 3)
true
pact> (<= 5.24 2.52)
false
pact> (<= "abc" "def")
true
```

6.4.9 =

x <a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset,guard,module{}]> *y* <a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset,guard,module{}]> → bool

Compare alike terms for equality, returning TRUE if X is equal to Y. Equality comparisons will fail immediately on type mismatch, or if types are not value types.

```
pact> (= [1 2 3] [1 2 3])
true
pact> (= 'foo "foo")
true
pact> (= { 'a: 2 } { 'a: 2})
true
```

6.4.10 >

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X > Y.

```
pact> (> 1 3)
false
pact> (> 5.24 2.52)
true
pact> (> "abc" "def")
false
```

6.4.11 >=

x <a[integer,decimal,string,time]> *y* <a[integer,decimal,string,time]> → bool

True if X >= Y.

```
pact> (>= 1 3)
false
pact> (>= 5.24 2.52)
true
pact> (>= "abc" "def")
false
```

6.4.12 ^

x <a[integer,decimal]> *y* <a[integer,decimal]> → <a[integer,decimal]>

x <a[integer,decimal]> *y* <b[integer,decimal]> → decimal

Raise X to Y power.

```
pact> (^ 2 3)
8
```

6.4.13 abs

x `decimal` → `decimal`

x `integer` → `integer`

Absolute value of X.

```
pact> (abs (- 10 23))
13
```

6.4.14 and

x `bool` *y* `bool` → `bool`

Boolean logic with short-circuit.

```
pact> (and true false)
false
```

6.4.15 and? {#and?}

a `x:<r> -> bool` *b* `x:<r> -> bool` *value* `<r>` → `bool`

Apply logical 'and' to the results of applying VALUE to A and B, with short-circuit.

```
pact> (and? (> 20) (> 10) 15)
false
```

6.4.16 ceiling

x `decimal` *prec* `integer` → `decimal`

x `decimal` → `integer`

Rounds up value of decimal X as integer, or to PREC precision as decimal.

```
pact> (ceiling 3.5)
4
pact> (ceiling 100.15234 2)
100.16
```

6.4.17 exp

x `<a[integer,decimal]>` → `<a[integer,decimal]>`

Exp of X.

```
pact> (round (exp 3) 6)
20.085537
```

6.4.18 floor

x `decimal` *prec* `integer` → `decimal`

x `decimal` → `integer`

Rounds down value of decimal X as integer, or to PREC precision as decimal.

```
pact> (floor 3.5)
3
pact> (floor 100.15234 2)
100.15
```

6.4.19 ln

x `<a[integer,decimal]>` → `<a[integer,decimal]>`

Natural log of X.

```
pact> (round (ln 60) 6)
4.094345
```

6.4.20 log

x `<a[integer,decimal]>` *y* `<a[integer,decimal]>` → `<a[integer,decimal]>`

x `<a[integer,decimal]>` *y* `<b[integer,decimal]>` → `decimal`

Log of Y base X.

```
pact> (log 2 256)
8
```

6.4.21 mod

x `integer` *y* `integer` → `integer`

X modulo Y.

```
pact> (mod 13 8)
5
```

6.4.22 not

x `bool` → `bool`

Boolean not.

```
pact> (not (> 1 2))
true
```

6.4.23 not? {#not?}

app `x:<r> -> bool` *value* `<r>` → `bool`

Apply logical 'not' to the results of applying VALUE to APP.

```
pact> (not? (> 20) 15)
false
```

6.4.24 or

x `bool` *y* `bool` → `bool`

Boolean logic with short-circuit.

```
pact> (or true false)
true
```

6.4.25 or? {#or?}

a `x:<r> -> bool` *b* `x:<r> -> bool` *value* `<r>` → `bool`

Apply logical 'or' to the results of applying VALUE to A and B, with short-circuit.

```
pact> (or? (> 20) (> 10) 15)
true
```

6.4.26 round

x `decimal` *prec* `integer` → `decimal`

x `decimal` → `integer`

Performs Banker's rounding value of decimal X as integer, or to PREC precision as decimal.

```
pact> (round 3.5)
4
pact> (round 100.15234 2)
100.15
```

6.4.27 shift

x `integer` *y* `integer` → `integer`

Shift X Y bits left if Y is positive, or right by -Y bits otherwise. Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the x is negative and with 0 otherwise.

```
pact> (shift 255 8)
65280
pact> (shift 255 -1)
127
pact> (shift -255 8)
-65280
```

(continues on next page)

(continued from previous page)

```
pact> (shift -255 -1)
-128
```

6.4.28 sqrt

x `<a[integer,decimal]>` → `<a[integer,decimal]>`

Square root of X.

```
pact> (sqrt 25)
5.0
```

6.4.29 xor

x `integer` *y* `integer` → `integer`

Compute bitwise X xor Y.

```
pact> (xor 127 64)
63
pact> (xor 5 -7)
-4
```

6.4.30 | {#|}

x `integer` *y* `integer` → `integer`

Compute bitwise X or Y.

```
pact> (| 2 3)
3
pact> (| 5 -7)
-3
```

6.4.31 ~ {#~}

x `integer` → `integer`

Reverse all bits in X.

```
pact> (~ 15)
-16
```

6.5 Keysets

6.5.1 define-keyset

name `string` *keyset* `string` → `string`

name `string` → `string`

Define keyset as NAME with KEYSET, or if unspecified, read NAME from message payload as keyset, similarly to 'read-keyset'. If keyset NAME already exists, keyset will be enforced before updating to new value.

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

Top level only: this function will fail if used in module code.

6.5.2 enforce-keyset

guard `guard` → `bool`

keysetname `string` → `bool`

Execute GUARD, or defined keyset KEYSETNAME, to enforce desired predicate logic.

```
(enforce-keyset 'admin-keyset)
(enforce-keyset row-guard)
```

6.5.3 keys-2

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match at least 2 keys in keyset.

```
pact> (keys-2 3 1)
false
```

6.5.4 keys-all

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match all keys in keyset.

```
pact> (keys-all 3 3)
true
```

6.5.5 keys-any

count `integer` *matched* `integer` → `bool`

Keyset predicate function to match any (at least 1) key in keyset.

```
pact> (keys-any 10 1)
true
```

6.5.6 read-keyset

key `string` → `keyset`

Read KEY from message data body as keyset ({ "keys": KEYLIST, "pred": PREDFUN }). PREDFUN should resolve to a keys predicate.

```
(read-keyset "admin-keyset")
```

6.6 Capabilities

6.6.1 compose-capability

capability `-> bool` → `bool`

Specifies and requests grant of CAPABILITY which is an application of a 'defcap' production, only valid within a (distinct) 'defcap' body, as a way to compose CAPABILITY with the outer capability such that the scope of the containing 'with-capability' call will "import" this capability. Thus, a call to '(with-capability (OUTER-CAP) OUTER-BODY)', where the OUTER-CAP defcap calls '(compose-capability (INNER-CAP))', will result in INNER-CAP being granted in the scope of OUTER-BODY.

```
(compose-capability (TRANSFER src dest))
```

6.6.2 create-module-guard

name `string` → `guard`

Defines a guard by NAME that enforces the current module admin predicate.

6.6.3 create-pact-guard

name `string` → `guard`

Defines a guard predicate by NAME that captures the results of 'pact-id'. At enforcement time, the success condition is that at that time 'pact-id' must return the same value. In effect this ensures that the guard will only succeed within the multi-transaction identified by the pact id.

6.6.4 create-principal

guard `guard` → `string`

Create a principal which unambiguously identifies GUARD.

```
(create-principal (read-keyset 'keyset))
(create-principal (keyset-ref-guard 'keyset))
(create-principal (create-module-guard 'module-guard))
(create-principal (create-user-guard 'user-guard))
(create-principal (create-pact-guard 'pact-guard))
```

6.6.5 create-user-guard

closure `-> bool` → `guard`

Defines a custom guard CLOSURE whose arguments are strictly evaluated at definition time, to be supplied to indicated function at enforcement time.

6.6.6 emit-event

capability `-> bool` → `bool`

Emit CAPABILITY as event without evaluating body of capability. Fails if CAPABILITY is not @managed or @event.

```
(emit-event (TRANSFER "Bob" "Alice" 12.0))
```

6.6.7 enforce-guard

guard `guard` → `bool`

keysetname `string` → `bool`

Execute GUARD, or defined keyset KEYSETNAME, to enforce desired predicate logic.

```
(enforce-guard 'admin-keyset)
(enforce-guard row-guard)
```

6.6.8 install-capability

capability `-> bool` → `string`

Specifies, and provisions install of, a *managed* CAPABILITY, defined in a 'defcap' in which a '@managed' tag designates a single parameter to be managed by a specified function. After install, CAPABILITY must still be brought into scope using 'with-capability', at which time the 'manager function' is invoked to validate the request. The manager function is of type 'managed:

requested:

->

', where'

' indicates the type of the managed parameter, such that for '(defcap FOO (bar:string baz:integer) @managed baz FOO-mgr ...)', the manager function would be '(defun FOO-mgr:integer (managed:integer requested:integer) ...)'. Any capability matching the 'static' (non-managed) parameters will cause this function to be invoked with the current managed value and that of the requested capability. The function should perform whatever logic, presumably linear, to validate the request, and return the new managed value representing the 'balance' of the request. NOTE that signatures scoped to a managed capability cause the capability to be automatically provisioned for install similarly to one installed with this function.

```
(install-capability (PAY "alice" "bob" 10.0))
```

6.6.9 keyset-ref-guard

keyset-ref `string` → `guard`

Creates a guard for the keyset registered as KEYSET-REF with 'define-keyset'. Concrete keysets are themselves guard types; this function is specifically to store references alongside other guards in the database, etc.

6.6.10 require-capability

capability `-> bool` → `bool`

Specifies and tests for existing grant of CAPABILITY, failing if not found in environment.

```
(require-capability (TRANSFER src dest))
```

6.6.11 validate-principal

guard `guard` *principal* `string` → `bool`

Validate that PRINCIPAL unambiguously identifies GUARD.

```
(enforce (validate-principal (read-keyset 'keyset) account) "Invalid account ID")
```

6.6.12 with-capability

capability `-> bool` *body* `[*]` → `<a>`

Specifies and requests grant of *acquired* CAPABILITY which is an application of a 'defcap' production. Given the unique token specified by this application, ensure that the token is granted in the environment during execution of BODY. 'with-capability' can only be called in the same module that declares the corresponding 'defcap', otherwise module-admin rights are required. If token is not present, the CAPABILITY is evaluated, with successful completion resulting in the installation/granting of the token, which will then be revoked upon completion of BODY. Nested 'with-capability' calls for the same token will detect the presence of the token, and will not re-apply CAPABILITY, but simply execute BODY. 'with-capability' cannot be called from within an evaluating defcap. Acquire of a managed capability results in emission of the equivalent event.

```
(with-capability (UPDATE-USERS id) (update users id { salary: new-salary }))
```

6.7 SPV

6.7.1 verify-spv

type `string` *payload* `object:<in>` → `object:<out>`

Performs a platform-specific spv proof of type TYPE on PAYLOAD. The format of the PAYLOAD object depends on TYPE, as does the format of the return object. Platforms such as Chainweb will document the specific payload types and return values.

```
(verify-spv "TXOUT" (read-msg "proof"))
```

6.8 Commitments

6.8.1 decrypt-cc20p1305

ciphertext `string` *nonce* `string` *aad* `string` *mac* `string` *public-key* `string` *secret-key* `string` → `string`

Perform decryption of CIPHERTEXT using the CHACHA20-POLY1305 Authenticated Encryption with Associated Data (AEAD) construction described in IETF RFC 7539. CIPHERTEXT is an unpadded base64url string. NONCE is a 12-byte base64 string. AAD is base64 additional authentication data of any length. MAC is the "detached" base64 tag value for validating POLY1305 authentication. PUBLIC-KEY and SECRET-KEY are base-16 Curve25519 values to form the DH symmetric key.Result is unpadded base64URL.

```
(decrypt-cc20p1305 ciphertext nonce aad mac pubkey privkey)
```

6.8.2 validate-keypair

public `string` *secret* `string` → `bool`

Enforce that the Curve25519 keypair of (PUBLIC,SECRET) match. Key values are base-16 strings of length 32.

```
(validate-keypair pubkey privkey)
```

6.9 REPL-only functions

The following functions are loaded automatically into the interactive REPL, or within script files with a `.repl` extension. They are not available for blockchain-based execution.

6.9.1 begin-tx

→ `string`

name `string` → `string`

Begin transaction with optional NAME.

```
pact> (begin-tx "load module")
"Begin Tx 0: load module"
```

6.9.2 bench

exprs ∗ → `string`

Benchmark execution of EXPRS.

```
(bench (+ 1 2))
```

6.9.3 commit-tx

→ `string`

Commit transaction.

```
pact> (begin-tx) (commit-tx)
"Commit Tx 0"
```

6.9.4 continue-pact

step `integer` → `string`

step `integer` *rollback* `bool` → `string`

step `integer` *rollback* `bool` *pact-id* `string` → `string`

step `integer` *rollback* `bool` *pact-id* `string` *yielded* `object:<{y}>` → `string`

Continue previously-initiated pact identified STEP, optionally specifying ROLLBACK (default is false), PACT-ID of the pact to be continued (defaults to the pact initiated in the current transaction, if one is present), and YIELDED value to be read with 'resume' (if not specified, uses yield in most recent pact exec, if any).

```
(continue-pact 1)
(continue-pact 1 true)
(continue-pact 1 false "[pact-id-hash]"))
(continue-pact 2 1 false "[pact-id-hash]" { "rate": 0.9 })
```

6.9.5 env-chain-data

new-data `object:~{public-chain-data}` → `string`

Update existing entries of 'chain-data' with NEW-DATA, replacing those items only.

```
pact> (env-chain-data { "chain-id": "TestNet00/2", "block-height": 20 })
"Updated public metadata"
```

6.9.6 env-data

json `<a[integer,string,time,decimal,bool,[<l>],object:<{o}>,keyset]>` → `string`

Set transaction JSON data, either as encoded string, or as pact types coerced to JSON.

```
pact> (env-data { "keyset": { "keys": ["my-key" "admin-key"], "pred": "keys-any" } })
"Setting transaction data"
```

6.9.7 env-dynref

iface `module` *impl* `module{}` → `string`

→ `string`

Substitute module IMPL in any dynamic usages of IFACE in typechecking and analysis. With no arguments, remove all substitutions.

```
(env-dynref fungible-v2 coin)
```

6.9.8 env-enable-repl-natives

enable `bool` → `string`

Control whether REPL native functions are allowed in module code. When enabled, fixture functions like 'env-sigs' are allowed in module code.

```
pact> (env-enable-repl-natives true)
"Repl natives enabled"
```

6.9.9 env-entity

→ `string`

entity `string` → `string`

Set environment confidential ENTITY id, or unset with no argument.

```
(env-entity "my-org")
(env-entity)
```

6.9.10 env-events

clear `bool` → `[object:*]`

Retreive any accumulated events and optionally clear event state. Object returned has fields 'name' (fully-qualified event name), 'params' (event parameters), 'module-hash' (hash of emitting module).

```
(env-events true)
```

6.9.11 env-exec-config

flags `[string]` → `[string]`

→ `[string]`

Queries, or with arguments, sets execution config flags. Valid flags: ["AllowReadInLocal","DisableHistoryInTransactionalMode","DisableInlineMemCheck","DisableModuleInstall","DisablePact40","DisablePact420","Di

```
pact> (env-exec-config ['DisableHistoryInTransactionalMode]) (env-exec-config)
["DisableHistoryInTransactionalMode"]
```

6.9.12 env-gas

→ `integer`

gas `integer` → `string`

Query gas state, or set it to GAS. Note that certain plaforms may charge additional gas that is not captured by the interpreter gas model, such as an overall transaction-size cost.

```
pact> (env-gasmodel "table") (env-gaslimit 10) (env-gas 0) (map (+ 1) [1 2 3]) (env-
↪gas)
7
```

6.9.13 env-gaslimit

limit `integer` → `string`

Set environment gas limit to LIMIT.

6.9.14 env-gaslog

→ `string`

Enable and obtain gas logging. Bracket around the code whose gas logs you want to inspect.

```
pact> (env-gasmodel "table") (env-gaslimit 10) (env-gaslog) (map (+ 1) [1 2 3]) (env-
↪gaslog)
["TOTAL: 7" "map:GUnreduced: 4" "+:GUnreduced: 1" "+:GUnreduced: 1" "+:GUnreduced: 1"]
```

6.9.15 env-gasmodel

model `string` → `string`

→ `string`

model `string` *rate* `integer` → `string`

Update or query current gas model. With just MODEL, "table" is supported; with MODEL and RATE, 'fixed' is supported. With no args, output current model.

```
pact> (env-gasmodel)
"Current gas model is 'fixed 0': constant rate gas model with fixed rate 0"
pact> (env-gasmodel 'table)
"Set gas model to table-based cost model"
pact> (env-gasmodel 'fixed 1)
"Set gas model to constant rate gas model with fixed rate 1"
```

6.9.16 env-gasprice

price `decimal` → `string`

Set environment gas price to PRICE.

6.9.17 env-gasrate

rate `integer` → `string`

Update gas model to charge constant RATE.

6.9.18 env-hash

hash `string` → `string`

Set current transaction hash. HASH must be an unpadded base64-url encoded BLAKE2b 256-bit hash.

```
pact> (env-hash (hash "hello"))
"Set tx hash to Mk3PAn3UowqTLEQfNlol6GsXPe-kuOWJSCU0cbgbcs8"
```

6.9.19 env-keys

keys `[string]` → `string`

DEPRECATED in favor of 'set-sigs'. Set transaction signer KEYS. See 'env-sigs' for setting keys with associated capabilities.

```
pact> (env-keys ["my-key" "admin-key"])
"Setting transaction keys"
```

6.9.20 env-namespace-policy

allow-root `bool` *ns-policy-fun* `ns:string ns-admin:guard -> bool` → `string`

Install a managed namespace policy specifying ALLOW-ROOT and NS-POLICY-FUN.

```
(env-namespace-policy (my-ns-policy-fun))
```

6.9.21 env-sigs

sigs `[object:*]` → `string`

Set transaction signature keys and capabilities. SIGS is a list of objects with "key" specifying the signer key, and "caps" specifying a list of associated capabilities.

```
(env-sigs [{'key: "my-key", 'caps: [(accounts.USER_GUARD "my-account")]}, {'key:
↪"admin-key", 'caps: []}
```

6.9.22 expect

doc `string` *expected* `<a>` *actual* `<a>` → `string`

Evaluate ACTUAL and verify that it equals EXPECTED.

```
pact> (expect "Sanity prevails." 4 (+ 2 2))
"Expect: success: Sanity prevails."
```

6.9.23 expect-failure

doc `string` *exp* `<a>` → `string`

doc `string` *err* `string` *exp* `<a>` → `string`

Evaluate EXP and succeed only if it throws an error.

```
pact> (expect-failure "Enforce fails on false" (enforce false "Expected error"))
"Expect failure: success: Enforce fails on false"
pact> (expect-failure "Enforce fails with message" "Expected error" (enforce false
↪"Expected error"))
"Expect failure: success: Enforce fails with message"
```

6.9.24 expect-that

doc `string` *pred* `value:<a> -> bool` *exp* `<a>` → `string`

Evaluate EXP and succeed if value passes predicate PRED.

```
pact> (expect-that "addition" (< 2) (+ 1 2))
"Expect-that: success: addition"
pact> (expect-that "addition" (> 2) (+ 1 2))
"FAILURE: addition: did not satisfy (> 2) : 3:integer"
```

6.9.25 format-address

scheme `string` *public-key* `string` → `string`

Transform PUBLIC-KEY into an address (i.e. a Pact Runtime Public Key) depending on its SCHEME.

6.9.26 load

file `string` → `string`

file `string` *reset* `bool` → `string`

Load and evaluate FILE, resetting repl state beforehand if optional RESET is true.

```
(load "accounts.repl")
```

6.9.27 mock-spv

type `string` *payload* `object:*` *output* `object:*` → `string`

Mock a successful call to 'spv-verify' with TYPE and PAYLOAD to return OUTPUT.

```
(mock-spv "TXOUT" { 'proof: "a54f54de54c54d89e7f" } { 'amount: 10.0, 'account: "Dave",
↪ 'chainId: "1" })
```

6.9.28 pact-state

→ `object:*`

clear `bool` → `object:*`

Inspect state from most recent pact execution. Returns object with fields 'pactId': pact ID; 'yield': yield result or 'false' if none; 'step': executed step; 'executed': indicates if step was skipped because entity did not match. With CLEAR argument, erases pact from repl state.

```
(pact-state)
(pact-state true)
```

6.9.29 print

value `<a>` → `string`

Output VALUE to terminal as unquoted, unescaped text.

6.9.30 rollback-tx

→ `string`

Rollback transaction.

```
pact> (begin-tx "Third Act") (rollback-tx)
"Rollback Tx 0: Third Act"
```

6.9.31 sig-keyset

→ `keyset`

Convenience function to build a keyset from keys present in message signatures, using 'keys-all' as the predicate.

6.9.32 test-capability

capability `->` `bool` → `string`

Acquire (if unmanaged) or install (if managed) CAPABILITY. CAPABILITY and any composed capabilities are in scope for the rest of the transaction.

```
(test-capability (MY-CAP))
```

6.9.33 typecheck

module `string` → `string`

module `string` *debug* `bool` → `string`

Typecheck MODULE, optionally enabling DEBUG output.

6.9.34 verify

module `string` → `string`

Verify MODULE, checking that all properties hold.

6.9.35 with-applied-env

exec <a> → <a>

Evaluate EXEC with any pending environment changes applied. Normally, environment changes must execute at top-level for the change to take effect. This allows scoped application of non-toplevel environment changes.

```
pact> (let ((a 1)) (env-data { 'b: 1 }) (with-applied-env (+ a (read-integer 'b))))
2
```

CHAPTER 7

The Pact Property Checking System

7.1 What is it?

Pact comes equipped with the ability for smart contract authors to express and automatically check properties – or, specifications – of Pact programs.

The Pact property checking system is our response to the current environment of chaos and uncertainty in the smart contract programming world. Instead of requiring error-prone smart contract authors to try to imagine all possible ways an attacker could exploit their smart contract, we can allow them to prove their code can't be attacked, all without requiring a background in formal verification.

For example, for an arbitrarily complex Pact program, we might want to definitively prove that the program only allows "administrators" of the contract to modify the database – for all other users, we're guaranteed that the contract's logic permits read-only access to the DB. We can prove such a property *statically*, before any code is deployed to the blockchain.

Compared with conventional unit testing, wherein the behavior of a program is validated for a single combination of inputs and the author hopes this case generalizes to all inputs, the Pact property checking system *automatically* checks the code against all possible inputs, and therefore all possible execution paths.

Pact does this by allowing authors to specify *schema invariants* about columns in database tables, and to state and prove *properties* about functions with respect to the function's arguments and return values, keyset enforcement, database access, and use of `enforce`.

For those familiar, the Pact's properties correspond to the notion of "contracts" (note: this is different than "smart contracts"), and Pact's invariants correspond to a simplified initial step towards refinement types, from the world of formal verification.

For this initial release we don't yet support 100% of the Pact language, and the implementation of the property checker *itself* has not yet been formally verified, but this is only the first step. We're excited to continue broadening support for every possible Pact program, eventually prove correctness of the property checker, and continually enable authors to express ever more sophisticated properties about their smart contracts over time.

7.2 What do properties and schema invariants look like?

Here's an example of Pact's properties in action – we declare a property alongside the docstring of the function to which it corresponds. Note that the function delegates its implementation of keyset enforcement to another function, `enforce-admin`, and we don't need to be concerned about its internal details. Our property states that if the transaction submitted to the blockchain runs successfully, it must be the case that the transaction has the proper signatures to satisfy the keyset named `admins`:

```
(defun read-account (id)
  @doc   "Read data for account ID"
  @model [(property (authorized-by 'admins))]

  (enforce-admin)
  (read 'accounts id ['balance 'ccy 'amount]))
```

There's a set of square brackets around our property because Pact allows multiple properties to be defined simultaneously:

```
[p1 p2 p3 ...]
```

Next, we see an example of schema invariants. For any table with the following schema, if our property checker succeeds, we know that all possible code paths will always maintain the invariant that token balances are greater than zero:

```
(defschema tokens
  @doc   "token schema"
  @model [(invariant (> balance 0))]

  username:string
  balance:integer)
```

7.3 How does it work?

Pact's property checker works by realizing the language's semantics in an SMT ("Satisfiability Modulo Theories") solver – by building a formula for a program, and testing the validity of that formula. The SMT solver can prove that there is no possible assignment of values to variables which can falsify a provided proposition about some Pact code. Pact currently uses Microsoft's Z3 theorem prover to power its property checking system.

Such a formula is built from the combination of the functions in a Pact module, the properties provided for those functions, and invariants declared on schemas in the module.

For any function definition in a Pact module, any subsequent call to another function is inlined. Before any properties are tested, this inlined code must pass typechecking.

For schema invariants, the property checker takes an inductive approach: it assumes that the schema invariants *hold* for the data currently in the database, and *checks* that all functions in the module maintain those invariants for any possible DB modification.

7.4 How do you use it?

After supplying any desired invariant and property annotations in your module, property checking is run by invoking `verify`:

```
(verify 'module-name)
```

This will typecheck the code and, if that succeeds, check all invariants and properties.

7.5 Expressing properties

7.5.1 Arguments, return values, and standard arithmetic and comparison operators

In properties, we can refer to function arguments directly by their names, and return values can be referred to by the name `result`:

```
(defun negate:integer (x:integer)
  @doc   "negate a number"
  @model [(property (= result (* -1 x)))]

  (* x -1))
```

Here you can also see that the standard arithmetic operators on integers and decimals work as they do in normal Pact code.

We can also define properties in terms of the standard comparison operators:

```
(defun abs:integer (x:integer)
  @doc   "absolute value"
  @model [(property (>= result 0))]

  (if (< x 0)
    (negate x)
    x))
```

7.5.2 Boolean operators

In addition to the standard boolean operators `and`, `or`, and `not`, Pact's property checking language supports logical implication in the form of `when`, where `(when x y)` is equivalent to `(or (not x) y)`. Here we define three properties at once:

```
(defun negate:integer (x:integer)
  @doc   "negate a number"
  @model
    [(property (when (< x 0) (> result 0)))
     (property (when (> x 0) (< result 0)))
     (property (and
       (when (< x 0) (> result 0))
       (when (> x 0) (< result 0))))]

  (* x -1))
```

7.5.3 Transaction abort and success

By default, every property is predicated on the successful completion of the transaction which would contain an invocation of the function being tested. This means that properties like the following:

```
(defun ensured-positive:integer (val:integer)
  @doc   "halts when passed a non-positive number"
  @model [(property (!= result 0))]

  (enforce (> val 0) "val is not positive")
  val)
```

will pass due to the use of `enforce`.

At run-time on the blockchain, if an `enforce` call fails, the containing transaction is aborted. Because `properties` are only concerned with transactions that succeed, the necessary conditions to pass each `enforce` call are assumed.

However, in some cases it's useful to assert when the function must succeed or abort. To write this kind of assertion, instead of `property`, you can use `succeeds-when` or `fails-when`, for example:

```
(defun ensured-positive:bool (val:integer)
  @model [
    ; this succeeds exactly when val > 0, and fails exactly when val <= 0
    (succeeds-when (> val 0))
    (fails-when    (<= val 0))

    ; however, it's valid to assert something weaker
    (succeeds-when (> val 1000))
    (fails-when    (< val -1000))
    ]
  (enforce (> val 0)))
```

With this model, we're guaranteed that no transaction will ever run on the blockchain with a non-positive `val`.

We've now seen all three valid forms of model assertions – `property`, `succeeds-when`, and `fails-when`.

7.5.4 More comprehensive properties API documentation

For the full listing of functionality available in properties, see the API documentation at Property and Invariant Functions.

7.6 Expressing schema invariants

Schema invariants are described by a more restricted subset of the functionality available in property definitions – effectively the functions which are not concerned with authorization, DB access, transaction success/failure, and function arguments and return values. See the API documentation at Property and Invariant Functions for the full listing of functions available in invariant definitions.

7.6.1 Keyset Authorization

In Pact, keys can be referred to by predefined names (defined by `define-keyset`) or passed around as values. The property checking system supports both styles of working with keysets.

For named keysets, the property `authorized-by` holds only if every possible code path enforces the keyset:

```
(defun admins-only (action:string)
  @doc   "Only admins or super-admins can call this function successfully.
  @model
```

(continues on next page)

(continued from previous page)

```
    [(property (or (authorized-by 'admins) (authorized-by 'super-admins)))
     (property (when (== "create" action) (authorized-by 'super-admins)))]

  (if (= action "create")
    (create)
    (if (= action "update")
      (update)
      (incorrect-action action))))
```

For the common pattern of row-level keyset enforcement, wherein a table might contain a row for each user, and each user's row contains a keyset that is authorized when the row is modified, we can ensure this pattern has been implemented correctly by using the `row-enforced` property.

For the following property to pass, the code must extract the keyset stored in the `ks` column in the `accounts` table for the row keyed by the variable `name`, and enforce it using `enforce-keyset`:

```
(row-enforced accounts 'ks name)
```

For some examples of `row-enforced` in action, see "A simple balance transfer example" and the section on "universal and existential quantification" below.

7.6.2 Database access

To describe database table access, the property language has the following properties:

- `(table-written accounts)` - that any cell of the table `accounts` is written
- `(table-read accounts)` - that any cell of the table `accounts` is read
- `(row-written accounts k)` - that the row keyed by the variable `k` is written
- `(row-read accounts k)` - that the row keyed by the variable `k` is read

For more details, see an example in "universal and existential quantification" below.

7.6.3 Mass conservation and column deltas

In some situations, it's desirable that the total sum of the values in a column remains the same before and after a transaction. Or to put it another way, that the sum of all updates to a column zeroes-out by the end of a transaction. To capture this pattern, we can express a "mass conservation" property using `column-delta`:

```
(= (column-delta accounts 'balance) 0.0)
```

This property asserts that the "column delta" is zero, where `column-delta` returns a numeric value of the sum of all changes to the column during the transaction.

For an example using this property, see "A simple balance transfer example" below.

We can also use `column-delta` to ensure that a column only ever increases during a transaction:

```
(>= 0 (column-delta accounts 'balance))
```

or that it increases by a set amount during a transaction:

```
(= 1 (column-delta accounts 'balance))
```

`column-delta` is defined in terms of the increase of the column from before to after the transaction (i.e. `after - before`) – not an absolute value of change. So here `1` means an increase of `1` to the column's total sum.

7.6.4 Universal and existential quantification

In examples like `(row-enforced accounts 'ks key)` or `(row-written accounts key)` above, we've so far only referred to function arguments by the use of the variable named `key`. But what if we wanted to talk about all possible rows that will be written, if a function doesn't simply update a single row?

In such a situation we could use universal quantification to talk about *any* such row:

```
(property
  (forall (key:string)
   (when (row-written accounts key)
     (row-enforced accounts 'ks key))))
```

This property says that for any possible row written by the function, the keyset in column `ks` must be enforced for that row.

Likewise instead of quantifying over all possible keys, if we wanted to state that there merely exists a row that is read during the transaction, we could use existential quantification like so:

```
(property
  (exists (key:string)
    (row-read accounts key)))
```

For both universal and existential quantification, note that a type annotation is required.

7.6.5 Defining and reusing properties

With `defproperty`, properties can be defined at the module level:

```
(module accounts 'admin-keyset
  @model
    [(defproperty conserves-mass
       (= (column-delta accounts 'balance) 0.0))
     (defproperty auth-required
       (authorized-by 'accounts-admin-keyset))]

  ; ...
  )
```

and then used at the function level by referring to the property's name:

```
(defun read-account (id)
  @model [(property auth-required)]

  ; ...
  )
```

7.7 A simple balance transfer example

Let's work through an example where we write a function to transfer some amount of a balance across two accounts for the given table:

```
(defschema account
  @doc "user accounts with balances"

  balance:integer
  ks:keyset)

(deftable accounts:{account})
```

The following code to transfer a balance between two accounts may look correct at first study, but it turns out that there are number of bugs which we can eradicate with the help of another property, and by adding an invariant to the table.

```
(defun transfer (from:string to:string amount:integer)
  @doc   "Transfer money between accounts"
  @model [(property (row-enforced accounts 'ks from))]

  (with-read accounts from { 'balance := from-bal, 'ks := from-ks }
    (with-read accounts to { 'balance := to-bal }
      (enforce-keyset from-ks)
      (enforce (>= from-bal amount) "Insufficient Funds")
      (update accounts from { "balance": (- from-bal amount) })
      (update accounts to   { "balance": (+ to-bal amount) }))))
```

Let's start by adding an invariant that balances can never drop below zero:

```
(defschema account
  @doc   "user accounts with balances"
  @model [(invariant (>= balance 0))]

  balance:integer
  ks:keyset)
```

Now, when we use `verify` to check all properties in this module, Pact's property checker points out that it's able to falsify the positive balance invariant by passing in an `amount` of `-1` (when the balance is `0`). In this case it's actually possible for the "sender" to steal money from anyone else by tranferring a negative amount! Let's fix that by enforcing `(> amount 0)`, and try again:

```
(defun transfer (from:string to:string amount:integer)
  @doc   "Transfer money between accounts"
  @model [(property (row-enforced accounts 'ks from))]

  (with-read accounts from { 'balance := from-bal, 'ks := from-ks }
    (with-read accounts to { 'balance := to-bal }
      (enforce-keyset from-ks)
      (enforce (>= from-bal amount) "Insufficient Funds")
      (enforce (> amount 0)         "Non-positive amount")
      (update accounts from { "balance": (- from-bal amount) })
      (update accounts to   { "balance": (+ to-bal amount) }))))
```

The property checker validates the code at this point, but let's add another property `conserves-mass` to ensure that it's not possible for the function to be used to create or destroy any money. We define it within `@model` at the module level:

```
(defproperty conserves-mass
  (= (column-delta accounts 'balance) 0.0))
```

And then we can use it within `@model` at the function level:

```
(defun transfer (from:string to:string amount:integer)
  @doc   "Transfer money between accounts"
  @model
    [(property (row-enforced accounts 'ks from))
     (property conserves-mass)]

  (with-read accounts from { 'balance := from-bal, 'ks := from-ks }
    (with-read accounts to { 'balance := to-bal }
      (enforce-keyset from-ks)
      (enforce (>= from-bal amount) "Insufficient Funds")
      (enforce (> amount 0)         "Non-positive amount")
      (update accounts from { "balance": (- from-bal amount) })
      (update accounts to   { "balance": (+ to-bal amount) }))))
```

When we run `verify` this time, the property checker finds a bug again – it's able to falsify the property when `from` and `to` are set to the same account. When this is the case, we see that the code actually creates money out of thin air!

To see how, let's focus on the two `update` calls, where `from` and `to` are set to the same value, and `from-bal` and `to-bal` are also set to what we'll call `previous-balance`:

```
(update accounts "alice" { "balance": (- previous-balance amount) })
(update accounts "alice" { "balance": (+ previous-balance amount) })
```

In this scenario, we can see that the second `update` call will completely overwrite the first one, with the value `(+ previous-balance amount)`. Alice has effectively created `amount` tokens for free!

We can fix this by adding another `enforce` (with `(!= from to)`) to prevent this unintended behavior:

```
(defun transfer (from:string to:string amount:integer)
  @doc   "Transfer money between accounts"
  @model
    [(property (row-enforced accounts 'ks from))
     (property conserves-mass)]

  (with-read accounts from { 'balance := from-bal, 'ks := from-ks }
    (with-read accounts to { 'balance := to-bal }
      (enforce-keyset from-ks)
      (enforce (>= from-bal amount) "Insufficient Funds")
      (enforce (> amount 0)         "Non-positive amount")
      (enforce (!= from to)         "Sender is the recipient")
      (update accounts from { "balance": (- from-bal amount) })
      (update accounts to   { "balance": (+ to-bal amount) }))))
```

And now we see that finally the property checker verifies that all of the following are true:

- the sender must be authorized to transfer money,
- it's not possible for a balance to drop below zero, and
- it's not possible for money to be created or destroyed.

CHAPTER 8

Property and Invariant Functions

These are functions available in properties and invariants – not necessarily in executable Pact code. All of these functions are available in properties, but only a subset are available in invariants. As a general rule, invariants have vocabulary for talking about the shape of data, whereas properties also add vocabulary for talking about function inputs and outputs, and database interactions. Each function also explicitly says whether it's available in just properties, or invariants as well.

8.1 Numerical operators

8.1.1 +

```
(+ x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Addition of integers and decimals.

Supported in either invariants or properties.

8.1.2 -

```
(- x y)
```

- takes `x`: *a*
- takes `y`: *a*

- produces *a*
- where *a* is of type `integer` or `decimal`

Subtraction of integers and decimals.

Supported in either invariants or properties.

8.1.3 *

```
(* x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Multiplication of integers and decimals.

Supported in either invariants or properties.

8.1.4 /

```
(/ x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Division of integers and decimals.

Supported in either invariants or properties.

8.1.5 ^

```
(^ x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Exponentiation of integers and decimals.

Supported in either invariants or properties.

8.1.6 log

```
(log b x)
```

- takes `b`: *a*
- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Logarithm of `x` base `b`.

Supported in either invariants or properties.

8.1.7 -

```
(- x)
```

- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Negation of integers and decimals.

Supported in either invariants or properties.

8.1.8 sqrt

```
(sqrt x)
```

- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Square root of integers and decimals.

Supported in either invariants or properties.

8.1.9 ln

```
(ln x)
```

- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Logarithm of integers and decimals base e.

Supported in either invariants or properties.

8.1.10 exp

```
(exp x)
```

- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Exponential of integers and decimals. e raised to the integer or decimal `x`.

Supported in either invariants or properties.

8.1.11 abs

```
(abs x)
```

- takes `x`: *a*
- produces *a*
- where *a* is of type `integer` or `decimal`

Absolute value of integers and decimals.

Supported in either invariants or properties.

8.1.12 round

```
(round x)
```

- takes `x`: `decimal`
- produces `integer`

```
(round x prec)
```

- takes `x`: `decimal`
- takes `prec`: `integer`
- produces `integer`

Banker's rounding value of decimal `x` as integer, or to `prec` precision as decimal.

Supported in either invariants or properties.

8.1.13 ceiling

```
(ceiling x)
```

- takes `x`: `decimal`
- produces `integer`

```
(ceiling x prec)
```

- takes `x`: `decimal`
- takes `prec`: `integer`
- produces `integer`

Rounds the decimal `x` up to the next integer, or to `prec` precision as decimal.

Supported in either invariants or properties.

8.1.14 floor

```
(floor x)
```

- takes `x`: `decimal`
- produces `integer`

```
(floor x prec)
```

- takes `x`: `decimal`
- takes `prec`: `integer`
- produces `integer`

Rounds the decimal `x` down to the previous integer, or to `prec` precision as decimal.

Supported in either invariants or properties.

8.1.15 mod

```
(mod x y)
```

- takes `x`: `integer`
- takes `y`: `integer`
- produces `integer`

Integer modulus

Supported in either invariants or properties.

8.2 Bitwise operators

8.2.1 &

```
(& x y)
```

- takes `x`: `integer`
- takes `y`: `integer`
- produces `integer`

Bitwise and

Supported in either invariants or properties.

8.2.2 |

```
(| x y)
```

- takes x: integer
- takes y: integer
- produces integer

Bitwise or

Supported in either invariants or properties.

8.2.3 xor

```
(xor x y)
```

- takes x: integer
- takes y: integer
- produces integer

Bitwise exclusive-or

Supported in either invariants or properties.

8.2.4 shift

```
(shift x y)
```

- takes x: integer
- takes y: integer
- produces integer

Shift x y bits left if y is positive, or right by -y bits otherwise.

Supported in either invariants or properties.

8.2.5 ~

```
(~ x)
```

- takes x: integer
- produces integer

Reverse all bits in x

Supported in either invariants or properties.

8.3 Logical operators

8.3.1 >

```
(> x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces `bool`
- where *a* is of type `integer` or `decimal`

True if `x` > `y`

Supported in either invariants or properties.

8.3.2 <

```
(< x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces `bool`
- where *a* is of type `integer` or `decimal`

True if `x` < `y`

Supported in either invariants or properties.

8.3.3 >=

```
(>= x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces `bool`
- where *a* is of type `integer` or `decimal`

True if `x` >= `y`

Supported in either invariants or properties.

8.3.4 <=

```
(<= x y)
```

- takes `x`: *a*
- takes `y`: *a*

- produces `bool`
- where *a* is of type `integer` or `decimal`

True if `x` <= `y`

Supported in either invariants or properties.

8.3.5 =

```
(= x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces `bool`
- where *a* is of type `integer`, `decimal`, `string`, `time`, `bool`, `object`, or `keyset`

True if `x` = `y`

Supported in either invariants or properties.

8.3.6 !=

```
(!= x y)
```

- takes `x`: *a*
- takes `y`: *a*
- produces `bool`
- where *a* is of type `integer`, `decimal`, `string`, `time`, `bool`, `object`, or `keyset`

True if `x` != `y`

Supported in either invariants or properties.

8.3.7 and

```
(and x y)
```

- takes `x`: `bool`
- takes `y`: `bool`
- produces `bool`

Short-circuiting logical conjunction

Supported in either invariants or properties.

8.3.8 or

```
(or x y)
```

- takes `x`: `bool`
- takes `y`: `bool`
- produces `bool`

Short-circuiting logical disjunction

Supported in either invariants or properties.

8.3.9 not

```
(not x)
```

- takes `x`: `bool`
- produces `bool`

Logical negation

Supported in either invariants or properties.

8.3.10 when

```
(when x y)
```

- takes `x`: `bool`
- takes `y`: `bool`
- produces `bool`

Logical implication. Equivalent to `(or (not x) y)`.

Supported in either invariants or properties.

8.3.11 and?

```
(and? f g a)
```

- takes `f`: *a* -> `bool`
- takes `g`: *a* -> `bool`
- takes `a`: *a*
- produces `bool`

`and` the results of applying both `f` and `g` to `a`

Supported in either invariants or properties.

8.3.12 or?

```
(or? f g a)
```

- takes `f`: *a* -> `bool`
- takes `g`: *a* -> `bool`
- takes `a`: *a*
- produces `bool`

`or` the results of applying both `f` and `g` to `a`

Supported in either invariants or properties.

8.4 Object operators

8.4.1 at

```
(at k o)
```

- takes `k`: `string`
- takes `o`: `object`
- produces *a*

```
(at i l)
```

- takes `i`: `integer`
- takes `o`: `list`
- produces `bool`

projection

Supported in either invariants or properties.

8.4.2 +

```
(+ x y)
```

- takes `x`: `object`
- takes `y`: `object`
- produces `object`

Object merge

Supported in either invariants or properties.

8.4.3 drop

```
(drop keys o)
```

- takes `keys`: [`string`]
- takes `o`: `object`
- produces `object`

drop entries having the specified keys from an object

Supported in either invariants or properties.

8.4.4 take

```
(take keys o)
```

- takes `keys`: [`string`]
- takes `o`: `object`
- produces `object`

take entries having the specified keys from an object

Supported in either invariants or properties.

8.4.5 length

```
(length o)
```

- takes `o`: `object`
- produces `integer`

the number of key-value pairs in the object

Supported in either invariants or properties.

8.5 List operators

8.5.1 at

```
(at k l)
```

- takes `k`: `string`
- takes `l`: [*a*]
- produces *a*

```
(at i l)
```

- takes `i`: `integer`
- takes `o`: `list`

- produces `bool`

projection

Supported in either invariants or properties.

8.5.2 length

```
(length s)
```

- takes `s`: [*a*]
- produces `integer`

List length

Supported in either invariants or properties.

8.5.3 contains

```
(contains x xs)
```

- takes `x`: *a*
- takes `xs`: [*a*]
- produces `bool`

```
(contains k o)
```

- takes `k`: `string`
- takes `o`: `object`
- produces `bool`

```
(contains value string)
```

- takes `value`: `string`
- takes `string`: `string`
- produces `bool`

List / string / object contains

Supported in either invariants or properties.

8.5.4 reverse

```
(reverse xs)
```

- takes `xs`: [*a*]
- produces [*a*]

reverse a list of values

Supported in either invariants or properties.

8.5.5 sort

```
(sort xs)
```

- takes `xs`: [*a*]
- produces [*a*]

sort a list of values

Supported in either invariants or properties.

8.5.6 drop

```
(drop n xs)
```

- takes `n`: `integer`
- takes `xs`: [*a*]
- produces [*a*]

drop the first `n` values from the beginning of a list (or the end if `n` is negative)

Supported in either invariants or properties.

8.5.7 take

```
(take n xs)
```

- takes `n`: `integer`
- takes `xs`: [*a*]
- produces [*a*]

take the first `n` values from `xs` (taken from the end if `n` is negative)

Supported in either invariants or properties.

8.5.8 make-list

```
(make-list n a)
```

- takes `n`: `integer`
- takes `a`: *a*
- produces [*a*]

create a new list with `n` copies of `a`

Supported in either invariants or properties.

8.5.9 map

```
(map f as)
```

- takes `f`: *a* -> *b*
- takes `as`: [*a*]
- produces [*b*]

apply `f` to each element in a list

Supported in either invariants or properties.

8.5.10 filter

```
(filter f as)
```

- takes `f`: *a* -> `bool`
- takes `as`: [*a*]
- produces [*a*]

filter a list by keeping the values for which `f` returns `true`

Supported in either invariants or properties.

8.5.11 fold

```
(fold f a bs)
```

- takes `f`: *a* -> *b* -> *a*
- takes `a`: *a*
- takes `bs`: [*b*]
- produces [*a*]

reduce a list by applying `f` to each element and the previous result

Supported in either invariants or properties.

8.6 String operators

8.6.1 length

```
(length s)
```

- takes `s`: `string`
- produces `integer`

String length

Supported in either invariants or properties.

8.6.2 +

```
(+ s t)
```

- takes s: string
- takes t: string
- produces string

```
(+ s t)
```

- takes s: [*a*]
- takes t: [*a*]
- produces [*a*]

String / list concatenation

Supported in either invariants or properties.

8.6.3 str-to-int

```
(str-to-int s)
```

- takes s: string
- produces integer

```
(str-to-int b s)
```

- takes b: integer
- takes s: string
- produces integer

String to integer conversion

Supported in either invariants or properties.

8.6.4 take

```
(take n s)
```

- takes n: integer
- takes s: string
- produces string

take the first n values from xs (taken from the end if n is negative)

Supported in either invariants or properties.

8.6.5 drop

```
(drop n s)
```

- takes `n`: `integer`
- takes `s`: `string`
- produces `string`

drop the first `n` values from `xs` (dropped from the end if `n` is negative)

Supported in either invariants or properties.

8.7 Temporal operators

8.7.1 add-time

```
(add-time t s)
```

- takes `t`: `time`
- takes `s`: *a*
- produces `time`
- where *a* is of type `integer` or `decimal`

Add seconds to a time

Supported in either invariants or properties.

8.8 Quantification operators

8.8.1 forall

```
(forall (x:string) y)
```

- binds `x`: *a*
- takes `y`: *r*
- produces *r*
- where *a* is *any type*
- where *r* is *any type*

Bind a universally-quantified variable

Supported in properties only.

8.8.2 exists

```
(exists (x:string) y)
```

- binds `x`: *a*
- takes `y`: *r*
- produces *r*
- where *a* is *any type*
- where *r* is *any type*

Bind an existentially-quantified variable

Supported in properties only.

8.8.3 column-of

```
(column-of t)
```

- takes `t`: `table`
- produces `type`

The *type* of `columns` for a given `table`. Commonly used in conjunction with quantification; e.g.: `(exists (col:(column-of accounts)) (column-written accounts col))`.

Supported in properties only.

8.9 Transactional operators

8.9.1 abort

```
abort
```

- of type `bool`

Whether the transaction aborts. This function is only useful when expressing propositions that do not assume transaction success. Propositions defined via `property` implicitly assume transaction success. We will be adding a new mode in which to use this feature in the future – please let us know if you need this functionality.

Supported in properties only.

8.9.2 success

```
success
```

- of type `bool`

Whether the transaction succeeds. This function is only useful when expressing propositions that do not assume transaction success. Propositions defined via `property` implicitly assume transaction success. We will be adding a new mode in which to use this feature in the future – please let us know if you need this functionality.

Supported in properties only.

8.9.3 governance-passes

```
governance-passes
```

- of type `bool`

Whether the governance predicate passes. For keyset-based governance, this is the same as something like `(authorized-by 'governance-ks-name)`. Pact's property checking system currently does not analyze the body of a capability when it is used for governance due to challenges around capabilities making DB modifications – the system currently assumes that a capability-based governance predicate is equally capable of succeeding or failing. This feature allows describing the scenarios where the predicate passes or fails.

Supported in properties only.

8.9.4 result

```
result
```

- of type *r*
- where *r* is *any type*

The return value of the function under test

Supported in properties only.

8.10 Database operators

8.10.1 table-written

```
(table-written t)
```

- takes `t`: *a*
- produces `bool`
- where *a* is of type `table` or `string`

Whether a table is written in the function under analysis

Supported in properties only.

8.10.2 table-read

```
(table-read t)
```

- takes `t`: *a*
- produces `bool`
- where *a* is of type `table` or `string`

Whether a table is read in the function under analysis

Supported in properties only.

8.10.3 cell-delta

```
(cell-delta t c r)
```

- takes `t`: *a*
- takes `c`: *b*
- takes `r`: `string`
- produces *c*
- where *a* is of type `table` or `string`
- where *b* is of type `column` or `string`
- where *c* is of type `integer` or `decimal`

The difference in a cell's value before and after the transaction

Supported in properties only.

8.10.4 column-delta

```
(column-delta t c)
```

- takes `t`: *a*
- takes `c`: *b*
- produces *c*
- where *a* is of type `table` or `string`
- where *b* is of type `column` or `string`
- where *c* is of type `integer` or `decimal`

The difference in a column's total summed value before and after the transaction

Supported in properties only.

8.10.5 column-written

```
(column-written t c)
```

- takes `t`: *a*
- takes `c`: *b*
- produces `bool`
- where *a* is of type `table` or `string`
- where *b* is of type `column` or `string`

Whether a column is written to in a transaction

Supported in properties only.

8.10.6 column-read

```
(column-read t c)
```

- takes `t`: *a*
- takes `c`: *b*
- produces `bool`
- where *a* is of type `table` or `string`
- where *b* is of type `column` or `string`

Whether a column is read from in a transaction

Supported in properties only.

8.10.7 row-read

```
(row-read t r)
```

- takes `t`: *a*
- takes `r`: `string`
- produces `bool`
- where *a* is of type `table` or `string`

Whether a row is read in the function under analysis

Supported in properties only.

8.10.8 row-written

```
(row-written t r)
```

- takes `t`: *a*
- takes `r`: `string`
- produces `bool`
- where *a* is of type `table` or `string`

Whether a row is written in the function under analysis

Supported in properties only.

8.10.9 row-read-count

```
(row-read-count t r)
```

- takes `t`: *a*
- takes `r`: `string`
- produces `integer`

- where *a* is of type `table` or `string`

The number of times a row is read during a transaction

Supported in properties only.

8.10.10 row-write-count

```
(row-write-count t r)
```

- takes `t`: *a*
- takes `r`: `string`
- produces `integer`
- where *a* is of type `table` or `string`

The number of times a row is written during a transaction

Supported in properties only.

8.10.11 row-exists

```
(row-exists t r time)
```

- takes `t`: *a*
- takes `r`: `string`
- takes `time`: one of {"before","after"}
- produces `bool`
- where *a* is of type `table` or `string`

Whether a row exists before or after a transaction

Supported in properties only.

8.10.12 read

```
(read t r)
```

- takes `t`: *a*
- takes `r`: `string`
- takes `time`: one of {"before","after"}
- produces `object`
- where *a* is of type `table` or `string`

The value of a read before or after a transaction

Supported in properties only.

8.11 Authorization operators

8.11.1 authorized-by

```
(authorized-by k)
```

- takes `k`: `string`
- produces `bool`

Whether the named keyset/guard is satisfied by the executing transaction

Supported in properties only.

8.11.2 row-enforced

```
(row-enforced t c r)
```

- takes `t`: *a*
- takes `c`: *b*
- takes `r`: `string`
- produces `bool`
- where *a* is of type `table` or `string`
- where *b* is of type `column` or `string`

Whether the keyset in the row is enforced by the function under analysis

Supported in properties only.

8.12 Function operators

8.12.1 identity

```
(identity a)
```

- takes `a`: *a*
- produces *a*
- where *a* is of type `table` or `string`

identity returns its argument unchanged

Supported in either invariants or properties.

8.12.2 constantly

```
(constantly a)
```

- takes `a`: *a*

- takes `b`: *b*
- produces *a*

constantly returns its first argument, ignoring the second

Supported in either invariants or properties.

8.12.3 compose

```
(compose f g)
```

- takes `f`: *a* -> *b*
- takes `g`: *b* -> *c*
- produces *c*

compose two functions

Supported in either invariants or properties.

8.13 Other operators

8.13.1 where

```
(where field f obj)
```

- takes `field`: `string`
- takes `f`: *a* -> `bool`
- takes `obj`: `object`
- produces `bool`

utility for use in `filter` and `select` applying `f` to `field` in `obj`

Supported in either invariants or properties.

8.13.2 typeof

```
(typeof a)
```

- takes `a`: *a*
- produces `string`

return the type of `a` as a string

Supported in either invariants or properties.