Qeditas Technical Documentation

The Qeditas Developers

October 3, 2016

Contents

Chapter 1

Introduction

This document is intended as a reference for those wanting to understand, modify or extend the code supporting Qeditas [31]. The document is currently under construction (in November 2015), as is the Qeditas code itself.

Qeditas is intended to be a realization of the QED Project [1] to construct a library of formalized mathematics. For those wanting to learn more about formalized mathematics [2] is a good starting point. Some popular systems for formalizing mathematics include Mizar [28], Isabelle [22], various HOL systems [14, 16], Coq [18] and Agda [24]. Some large formalizations of major mathematical results are described in [11], [12] and [15].

Qeditas uses a block chain and distributed consensus system to maintain both a library of formal publications and information about who made a definition or proved a theorem. In addition, the block chain maintains and enforces a rights management system determining conditions under which an object can be imported and used (without repeating the definition) and when a theorem can imported as something known (without needing to reprove it). The first network with a block chain and distributed consensus system was Bitcoin and Nakamoto's white paper [21] provides a good introduction. Qeditas uses a proof of stake consensus system [6] combined with proof of storage (similar to proof of retrievability [20]). Proof of stake is sensitive to the initial distribution, and Qeditas has an initial distribution based on a snapshot of the Bitcoin block chain. This makes Qeditas a Bitcoin "spin-off" [26, 10].

1.1 Code Repository

The main git repository is publicly available:

```
git clone git://qeditas.org/qeditas.git
```

There are three primary git branches: `master`, `dev` and `testing`. The `master` branch is intended to contain the code which can be tagged to create specific Qeditas versions. The `dev` branch is where code can be written,

modified and to some degree tested. The `testing` branch contains a number of unit tests. The code in `dev` should regularly be merged into `testing` and the unit tests run to ensure the unit tests still pass. In addition, new unit tests should be added as new code is added to the `dev` branch. Unfortunately, the `dev` branch has developed significantly since it was last merged into the `testing` branch, and so the code and unit tests in the `testing` branch is out of date with respect to the code in the `dev` and `master` branches.

When the code in `dev` is stable, it should be merged to `master`.

The source for this document is also part of the `dev` branch and the intention is for it to correspond to the code in the `dev` branch. Likewise, the source for this document can be merged into the `master` branch when appropriate.

Another branch, `initdistr`, contains code for computing the ledger tree for the initial distribution of Qeditas currency units. This distribution was based on a snapshot of the Bitcoin block chain.

1.2 Qeditas Theory in Coq

There is also a separate formal Qeditas related development in Coq in which certain properties were proven. The git repository is named `qeditastheory` and is also publicly available:

```
git clone git://qeditas.org/qeditastheory.git
```

The Coq code is somewhat out of date since some aspects of Qeditas have changed in the meantime. Nevertheless the Coq development should usually indicate how different data types were intended to be used and what properties certain functions were meant to have. In appropriate places we will point not only to the Qeditas OCaml code, but also corresponding code in the Coq version.

1.3 License and Credit

Qeditas is an open source project and all code and documentation is released under the MIT License. The code and documentation is attributed to "The Qeditas developers" rather than giving a list of names or handles. Some open source developers advise not to include names and handles inside code comments. Having a name in the code might suggest a kind of "ownership" that may make others uncomfortable making modifications to the code. If individual coders wish to record their contributions to the code, the appropriate place would be in this technical documentation. (For example, most of the initial code for Qeditas and the first version of this documentation was written by Bill White in 2015 with significant additions and modifications by Trent Russell in 2016.) Also, if code is taken or ported from other open source projects, this should be noted in this document. (For example, the code for elliptic curves and proof checking was taken from Egal [3] and this is noted in the appropriate sections.)

Chapter 2

Configuration Related Code

The modules config and setconfig are for customizing the configuration of Qeditas. The `configure` script creates an OCaml file `config.ml` setting default values for the variables exposed in the interface `config.mli`:

- datadir : the location of the main directory containing the local Qeditas configuration file, wallet file, and other data (usually `.qeditas` in the user's home directory)
- testnet : set to true if Qeditas is running on the testnet instead of the mainnet
- staking : set to true if Qeditas should stake
- ip : optionally the IP address to listen for incoming connections
- ipv6 : optionally the IPv6 address to listen for incoming connections
- port : the port to listen for incoming connections
- socks : None if connections are not routed through SOCKS; Some(v) if connections are routed through SOCKS protocol v where v is 4 or 5[1]
- maxconns : the maximum number of connections
- seed : the initial seed which is used to initialized the current stake modifier and future stake modifier.
- lastcheckpoint : the last checkpoint (currently unused)
- randomseed : an optional string used to seed the OCaml Random module. If randomseed is given it should be cryptographically strong and new each time Qeditas is started. If randomseed is not given, then Random is initialized using data from `/dev/random`. If `/dev/random` does not exist (e.g., under MS Windows), randomseed must be given and it is the user's responsibility to ensure randomseed is cryptographically strong and fresh.

[1] At the moment, 5 is not yet supported.

- checkpointskey : the private key for signing checkpoints (in Qeditas testnet WIF format). Signed checkpoints are only intended for the testnet, and only until the testnet is sufficiently stable. The corresponding public key is (x, y) where

$$x = 637172037326910029666274935234783955109256379641381851991015304296144946082 15$$
$$y = 145515389931093524325586496465640740010100594169115250324282127647820589012 34$$

  These values can be found in `blocktree.ml` as the settings for checkpointspubkeyx and checkpointspubkeyy. If a future version of Qeditas should use a different signing key for checkpoints, simply update checkpointspubkeyx and checkpointspubkeyy in `blocktree.ml` to the new public key values.

The functions exposed in the interface `setconfig.mli` override the default compiled settings by reading a configuration file and checking the command line arguments of `qeditasd` or `qeditascli`. This is done by calling datadir_from_command_line to set datadir from the command line if the argument `-datadir` was given, then calling process_config_file to read the `qeditas.conf` file in datadir, and finally calling process_config_args to set the remaining configuration variables by processing other command line arguments than `-datadir`.

Chapter 3

Utilities

The module utils (`utils.ml` and `utils.mli`) defines a few simple functions used by many other modules. The current code is for handling the log file, for computing the "era" and corresponding maximum block size, and for initializing and using OCaml's Random module. More details about each of these follow.

3.1 Logging

Qeditas logs information to a `log` file which should be in the main directory given by datadir (presumably `.qeditas`) or the `testnet` subdirectory. The function openlog and closelog open and close the log file. The relevant out_channel is the value held in the log ref cell.

(Note: At the moment Qeditas logs an excessive amount of information intended for debugging purposes.)

3.2 Eras and Block Size

The block height determines what "era" Qeditas is in, as computed by the era function. The era is used to determine the block rewards and the maximum block delta size. (A "block" consists of a "header" and a "delta.") The initial era is "Era 1" which lasts from blocks 1 through 70000 (when the reward is 25 fraenks and the maximum block delta size is 500,000 bytes). The next 41 eras ("Era 2" through "Era 42") each last 210000 blocks, at which time the reward halves and the maximum block delta size doubles. Starting at block 8680001 (in roughly 165 years after block one) the final era ("Era 43") begins. In this final era the block reward is 0 and the maximum block delta size is 512 megabytes.

The code for computing the maximum block delta size (maxblockdeltasize) is included in the utils module because it is needed in the net module to determine the maximum message size. On the other hand, the function for computing the reward (rewfn) is not given until the module block.

3.3 Randomness

The boolean random_initialized is set to false until initialize_random_seed has been called. initialize_random_seed is called the first time a random value is requested.[1]

The function initialize_random_seed calls Random.full_init using some data (or raises an exception). The data is either the value of randomseed or 32 bytes from `/dev/random`. If randomseed is not set and `/dev/random` does not exist, then a Failure exception is raised.

The functions rand_bit, rand_int32 and rand_int64 use OCaml's Random module to obtain a boolean, int32 or int64, respectively. A corresponding function rand_256 in the module sha256 generates a 256-bit random number using OCaml's Random module. (It is in this later module because it uses code to create a big_int from 8 int32 values.) There is also a (currently unused) function strong_rand_256 in the module sha256 which obtains the 256-bit random big_int using data from `/dev/random`.

[1] This way of doing things was so that, in principle, a user could use Qeditas in a way that does not require randomness (e.g., never signing a block or transaction and never generating a private key). In practice a random number is requried by initialize in `qeditas.ml` to set the node's nonce – this_nodes_nonce in net.

Chapter 4

Serialization

The module ser (`ser.ml` and `ser.mli`) contains the basic code for serializing data. Throughout the code there are functions with names of the form seo_τ (output) and sei_τ (input). In each case seo_τ is a function for creating a serialized output for an element of type τ and sei_τ is a function for creating an element of type τ given its serialization.

There are two representations for the serializations: strings and channels and the input and output functions are polymorphic so they can support both representations. The types seosbt and seist correspond to the string representation and there are corresponding atomic functions seosb (output bits to a string buffer), seosbf (flush output to string buffer) and seis (input bits from a string). The types seosct and seict correspond to the channel representation and there are corresponding atomic functions seoc (output bits to a channel), seocf (flush output channel) and seic (input bits from a channel).

The functions to output its take three arguments: the number of bits n, the bits to output as an integer m with $0 \leq m < 2^n$ and the serialization output object. The functions to flush the output takes the serialization output object and ensures any remaining its are output, assuming the remaining its are zero. The functions to input its from a channel take two arguments: the number of bits n to input and the serialization input object. It returns both an integer m where $0 \leq m < 2^n$ and the serialization input object.

The remainder of the functions defined in ser are for serialization basic data: booleans, bytes, 32-bit integers, 64-bit integers, big integers (the OCaml type big_int) assumed to be 256-bit integers and strings. There are also other serialization functions for integers: seo_varint and sei_varint uses the varint representation used in Bitcoin, while seo_varintb and sei_varintb uses a different compact representation to represent numbers less than $65,536$.

In addition there are functions used to construct serialization functions for list types, option types and product types with up to 6 components. For example, there are functions seo_list and sei_list. When serialization functions are needed for lists of bytes, we simply use seo_list applied to seo_int8 and sei_list applied to sei_int8.

Note that the serialization code is inherently higher-order (functions are first-class values). Firstly, the atomic functions are passed as arguments to the serialization functions for each type so the same serialization code can be used for both representations. Secondly, the serialization functions themselves are passed to functions like seo_list and sei_list.

There is one minor issue with the serialization code which may be confusing and hopefully will not be a nightmare to maintain. The bits are used to construct a byte from least significant to most significant. As a consequence, different ways to output the same sequence of bits can be confusing. Let o be an atomic output function (either seosb or seoc). For example, suppose we wish to output the bit 0 followed by the bit 1. We can do this in one of two ways:

- Call o twice: first as $o\,1\,0$ (to output the one bit 0) and then as $o\,1\,1$ (to output the one bit 1).
- Call o once as $o\,2\,2$ (to output the two bits 10, the binary representation of 2).

In some places this can make serialization code difficult to correctly interpret.

Note: The unit tests for the ser module are in `basicunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described above should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

Chapter 5

Cryptographic Hashing

The modules hashaux, sha256, ripemd160, hash and htree contain code for cryptographic hashing functions. The two hashing functions supported are `SHA256` [23] and `RIPEMD-160` [8]. The `RIPEMD-160` code only supports hashing a 256 bit input and is assumed to be called on the result of applying `SHA256`.

Profiling suggests that the hashing functions are the biggest computational bottleneck in Qeditas. Improvements to this code could make Qeditas run significantly faster.

Note: The unit tests for these modules are in `basicunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

5.1 Auxiliary Functions

The module hashaux implements a few helper functions needed by both hashing functions.

- hexsubstring_int32 takes a string of hexadecimal digits and a position. The 8 characters starting at the position are interpreted as a 32-bit integer (big endian).
- int32_hexstring takes a string buffer and a 32-bit integer and adds 8 hexadecimal digits to the buffer representing the integer (big endian).
- big_int_sub_int32 takes a big integer x and an integer i and returns the 32-bit integer resulting from shifting away i bits of x (i.e., dividing by 2^i) and then taking the 32 least significant bits (i.e., modulo 2^{32}).
- int32_big_int_bits takes a 32-bit integer x and an integer i and returns the big integer resulting from shifting x forward by i bits (i.e., multiplying by 2^i).

- int32_rev takes a 32-bit integer of the form $b_3 2^{24} + b_2 2^{16} + b_1 2^8 + b_0$ and returns the reversed 32-bit integer $b_0 2^{24} + b_1 2^{16} + b_2 2^8 + b_3$.

5.2 Sha256

The module sha256 defines a type md256 (message digest of 256 bits) as a product of 8 32-bit integers. (The type md256 is also sometimes used to represent other 256-bit numbers, such as the x or y component of a public key.) There is also an array currblock of 16 32-bit integers. Various other arrays are used internally and not exposed by the interface.

The following functions are defined:

- sha256init initializes the state to begin performing the hashing operation.
- sha256round performs one round of the hashing operation.
- getcurrmd256 returns the current md256 (extracted from the internal array currhashval).
- sha256str returns the result of hashing a given string with `SHA256`.
- sha256str returns the result of double hashing a given string with `SHA256`.
- md256_hexstring converts a 256-bit message digest to the corresponding hexadecimal string.
- hexstring_md256 converts a hexadecimal string to the 256-bit message digest to the corresponding hexadecimal string.
- md256_big_int converts a 256-bit message digest to the corresponding big integer.
- big_int_md256 converts a big integer (assuming it is less than 2^{256}) to the 256-bit message digest to the corresponding hexadecimal string.
- seo_md256 serializes a 256-bit message digest.
- sei_md256 deserializes a 256-bit message digest.

In addition, there are two functions for creating 256-bit random values of type big_int:

- strong_rand_256 : reads 256 bits from `/dev/random`. This function is currently unused.
- rand_256 : uses OCaml's Random module after initialized (see utils) to obtain 8 32-bit integers which are combined into a 256-bit big_int value.

5.3 Ripemd160

The module ripemd160 implements RIPEMD-160 restrict to 256-bit message digests as inputs. The module defines a type md (message digest of 160 bits) as a product of 5 32-bit integers.

The following functions are defined:

- ripemd160_md256 returns the result of hashing a given 256-bit message digest with RIPEMD-160.
- md_hexstring converts a 160-bit message digest to the corresponding hexadecimal string.
- hexstring_md converts a hexadecimal string to the 160-bit message digest to the corresponding hexadecimal string.

5.4 Hash

The module hash is important. It defines a type hashval as 5 32-bit integers (representing a 160-bit hash value).

A function hash160 takes an arbitrary string to the result of hashing first by SHA256 and then by RIPEMD-160. The type hashval is implemented the same way as the type md in the module ripemd160. If they were defined differently, the function hash160 would be ill-typed.

Note: The Coq formalization contains Coq module a CryptoHashes which corresponds to some of what is in the hash module. In particular, a type of hashval is defined along with functions to hash natural numbers, addresses (which are defined to be 160 bit sequences in the Coq module Addr) and pairs of hash values. These functions are injective and give disjoint hash values. From these, a number of other hashing functions are defined in ways that continue to ensure injectivity and disjointness. The Coq representation is idealized. Hash values is infinite and the hashing functions are not cryptographic hashing functions. For more information, see [30].

There are a variety of functions for creating, using and combining hash values. The following functions

- hashval_bitseq converts a hash value to a list of 160 booleans.
- hashval_hexstring converts a hash value to a string of 40 hexadecimal digits.
- hexstring_hashval converts a string of 40 hexadecimal digits to a hash value.
- printhashval prints a hash value as 40 hexadecimal digits.
- hashval_rev performs a bytewise reversal of the hash value.[1]
- hashval_big_int converts a hash value to a big integer.

[1] This seems to be unused.

- big_int_hashval converts a big integer to a hash value.
- seo_hashval serializes hash values.
- sei_hashval deserializes hash values.

The following functions create (effectively) unique hash values from given input. Internally in each case the value being hashed is prefixed with a distinct 32-bit integer so that the hash value given by different functions will be unique. For example, hashint32 prefixes the 32-bit integer with the 32-bit integer 1 while hashint64 prefixes the 64-bit integer with the 32-bit integer 2.

- hashint32 hashes a 32-bit integer.
- hashint64 hashes a 64-bit integer.
- hashpair hashes a pair of hashes.
- hashpubkey hashes a public key, given as two md256 values.
- hashpubkeyc hashes a compressed public key, given by a boolean (indicating if y is even or odd) and one md256 values (giving x).
- hashtag combines a hash value with a 32-bit integer to create a different hash value. This is used when we wish to ensure later data structures create unique hash values.
- hashlist hashes a list of hash values. This could be implemented by a simple recursion using hashpair, but this would be inefficient. Instead the list is iterated over with sha256round being called when appropriate.
- hashfold is given a function f which returns a hashval for a given input and a list of appropriate inputs for f and iteratively calls f on the components of the list while performing sha256round to compute a hash value for the list of hash values computed by f over the list.
- hashbitseq takes a list of booleans and creates a hash values. The naive way of doing this using hashlist would be too inefficient. Instead the booleans are treated as 32-bit integers by considering them in groups of 32.

Sometimes optional hash values are used. This is important, for example, when we want to have an "empty" hash value $\perp$ corresponding to the hash of some "empty" data.

- ohashlist takes a list of hash values and computes an optional hash value. The optional hash value is $\perp$ if and only if the input is the empty list.
- hashopair takes two optional hash values and returns an optional hash value. The output is $\perp$ if and only if both inputs were $\perp$.

- hashopair1 takes a hash value x and an optional hash value y and returns a hash value (known to not be $\perp$). hashopair1 is essentially the special case of hashopair where the first value is known not to be $\perp$.

- hashopair2 takes an optional hash value x and a hash value y and returns a hash value (known to not be $\perp$). hashopair2 is essentially the special case of hashopair where the second value is known not to be $\perp$.

In addition, various types of addresses are defined. Fundamentally there are four kinds of addresses: p2pkhaddr (pay to public key hash addresses, a.k.a., *p2pkh addresses*), p2shaddr (pay to script hash addresses, a.k.a., *p2sh addresses*), termaddr (*term addresses*) and pubaddr (*publication addresses*). Each of these types is defined the same way as hash values (as 5 32-bit integers) and so an object of one of these types can be used as an object of another.

- p2pkhaddr A pay to public key hash addresses is the hash value obtained by hashing a public key. The intention is that the holder of the corresponding private key can sign transactions related to the address. The code for checking such signatures is in the module signat.

- p2shaddr A pay to script hash address is the hash value obtained by hashing a script.[2] Such a script can act as a generalized signature in the following sense: the script is executed and if the result is 1 then the generalized signature is accepted. This is a "generalized signature" since some of the script operations check a signature. The code for executing scripts and checking generalized signatures is in the module script.

- termaddr Term addresses are hash values obtained in one of three ways:

  1. A term address may be the hash root of a closed simply typed term t. This is the global (theory independent) term address of t.
  2. Given a theory T and a closed term t which has type τ in the theory T, the combined hash of T, hash root of t and the hash of τ gives a term address.[3] This is the address of the term t in the theory T.
  3. Given a theory T and a closed proposition t, the combined hash of T and hashroot of t gives a term address.[4]

  Ownership information about a term or proposition (either globally or as part of a theory) is stored at corresponding term address. The author of the first document published which defines a term or proves a proposition can and must also supply ownership information. This ownership information determines the conditions under which the term or proposition

[2] Qeditas uses essentially the same scripting language as Bitcoin as of early 2015. Two Bitcoin operations are not supported: OP_SHA1 and OP_RIPEMD160.

[3] The combined hash is again hashed with a tag with 32 to avoid the possibility that the combined hash value is the same as a different kind of term address.

[4] The combined hash is again hashed with a tag with 33 to avoid the possibility that the combined hash value is the same as a different kind of term address.

can be imported into future documents. Term addresses corresponding to terms or propositions within a theory are also used to ensure terms have the correct type (without needing to repeat the definition) in the theory and to ensure propositions are already known (without needing to repeat a proof).

- pubaddr A publication address corresponds to the hash root of a published document, theory specification or signature specification.

In addition to the four basic kinds of addresses, there are two other types of addresses:

- payaddr The type payaddr (of *pay addresses*) is the disjoint sum of the types p2pkhaddr and p2shaddr. This is implemented by taking a boolean along with 8 32-bit integers. The 8 32-bit integers is a hash value representing either a p2pkhaddr or a p2pkhaddr. The boolean is false if the hash value represents a p2pkhaddr, and is true if the hash value represents a p2shaddr.

- addr The type addr (of addresses) is the disjoint sum of the four types p2pkhaddr, p2shaddr, termaddr and pubaddr. This is implemented by taking an integer $i \in \{0, 1, 2, 3\}$ along with 8 32-bit integers. The 8 32-bit integers is a hash value representing either a p2pkhaddr, a p2shaddr, a termaddr or a pubaddr. If $i = 0$, the hash value represents a p2pkhaddr. If $i = 1$, the hash value represents a p2shaddr. If $i = 2$, the hash value represents a termaddr. If $i = 3$, the hash value represents a pubaddr.

The following functions operate on addresses.

- hashval_p2pkh_payaddr gives the pay address corresponding to a hash value interpreted as a pay to public key hash addresses.

- hashval_p2sh_payaddr gives the pay address corresponding to a hash value interpreted as a pay to script hash address.

- hashval_p2pkh_addr gives the address corresponding to a hash value interpreted as a pay to public key hash addresses.

- hashval_p2sh_addr gives the address corresponding to a hash value interpreted as a pay to script hash address.

- hashval_term_addr gives the address corresponding to a hash value interpreted as a term address.

- hashval_pub_addr gives the address corresponding to a hash value interpreted as a publication address.

- addr_bitseq returns a list of 162 booleans corresponding to an address, where the first two booleans determine which kind of address it is and the remaining 160 are the hash value.

- bitseq_addr returns an address given a list of 162 booleans.

- p2pkhaddr_payaddr converts a pay to public key hash address (a hash value) to a pay address by indicating it is a pay to public key hash address.
- p2shaddr_payaddr converts a pay to public key hash address (a hash value) to a pay address by indicating it is a pay to script hash address.
- p2pkhaddr_addr converts a pay to public key hash address (a hash value) to an address by indicating it is a pay to public key hash address.
- p2shaddr_addr converts a pay to public key hash address (a hash value) to an address by indicating it is a pay to script hash address.
- payaddr_addr converts a pay address to an address. In practice this simply means converting the first component from a boolean to an integer (false to 0 and true to 1).
- termaddr_addr converts a term address (a hash value) to an address by indicating it is a term address.
- pubaddr_addr converts a publication address (a hash value) to an address by indicating it is a publication address.
- payaddr_p checks if an address is a pay address.
- p2pkhaddr_p checks if an address is a pay to public key hash address.
- p2shaddr_p checks if an address is a pay to script hash address.
- termaddr_p checks if an address is a term address.
- pubaddr_p checks if an address is a publication address.
- hashaddr hashes the address creating a unique hash value. (This is different from the underlying hash value of the address since the prefix is included before rehashing.)
- hashpayaddr performs the same operation of hashaddr but only on pay addresses.
- hashtermaddr performs the same operation of hashaddr on term addresses.
- hashpubaddr performs the same operation of hashaddr on publication addresses.
- seo_addr serializes an address.
- sei_addr deserializes an address.
- seo_payaddr serializes a pay address.
- sei_payaddr deserializes a pay address.
- seo_termaddr serializes a term address.

- sei_termaddr deserializes a term address.
- seo_pubaddr serializes a publication address.
- sei_pubaddr deserializes a publication address.

5.5 Hash Trees

The module htree defines a polymorphic type htree which stores data in a tree indexed by a list of booleans. In practice the list of booleans comes from a hash value. The following functions are exported:

- htree_lookup is given a boolean list and an htree and returns the data if found and None if it is not found.
- htree_create is given a boolean list $\overline{b}$ and data x and returns a new htree with only this single entry (x at position $\overline{b}$).
- htree_insert is given an htree, a boolean list and data x and returns the result of inserting the data into the htree (x at position $\overline{b}$). This will shadow data already at position $\overline{b}$ if there were any. (In practice this should never happen since $\overline{b}$ should be obtained from a hash value determined by x.)
- ohtree_hashroot computes an optional hash value given a function f (which computes optional hash values for members of the underlying type), the current depth n and an optional htree. This is essentially the Merkle root of the htree.

In practice htree is used in two ways: one is to story theories and the other is to store theory-specific signatures.[5] These specific uses will be discussed in Chapter 9 where the module mathdata is considered.

[5]Here *signature* is used to mean a collection of constants, definitions and known propositions and should not be confused with cryptographic signatures.

Chapter 6

Cryptocurrency Operations

The modules secp256k1, cryptocurr, signat and script contain code for cryptocurrency related operations. In particular, secp256k1 implements the elliptic curve `secp256k1` [27], cryptocurr supports base 58 formats for private keys and addresses, signat supports cryptographic signatures and script supports the Bitcoin scripting language (mostly).

Note: The unit tests for these modules are in `basicunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

6.1 Elliptic Curve Code

The module secp256k1 contains the operations for the corresponding elliptic curve [27]. Most of the code in this module was taken from the code for Egal [3]. 256-bit integers are represented using big integers (big_int) from OCaml's `nums` library. A function evenp is defined and exposed since it is used elsewhere.

The 256-bit prime p used in the elliptic curve is exposed as the big integer _p. The following functions implement operations modulo p:

- add implements addition modulo p.
- mul implements multiplication modulo p.
- pow implements x^i modulo p where x is a big integer and i is an integer.
- eea implements the Extended Euclidean Algorithm which is used to compute multiplicative inverses modulo p.

Points on the elliptic curve are represented by element of type pt which is defined to be an optional pair (x, y) of big integers. None represents the zero point (point at infinity). The following functions are defined and exposed:

- addp implements addition of two points.
- smulp implements scalar multiplication of a big integer to a point.

The base point g on the curve (which generates the group) is exposed as _g. The order n of g is exposed as the big integer _n. The function curve_y takes a boolean e and a big integer x and returns the big integer y such that (x, y) is a point on the curve where y is even if e is true and y is odd if e is false.

As usual, there are serialization functions seo_pt and sei_pt for points. The serialization functions assume the components x and y of the point are positive integers less than 2^{256}.

6.2 Cryptocurrency Operations

The module cryptocurr implements functions which convert private keys and addresses to and from base 58 representations.

- base58 converts a big integer into a base 58 string.
- frombase58 converts a base 58 string to a big integer.
- qedwif converts a big integer (private key) and a boolean (indicating if it is for a compressed address) to a base 58 string. The Qeditas WIF format uses a two byte prefix of $5, 8$ for compressed addresses and $2, 30$ for uncompressed addresses. The result is that compressed WIFs start with the character `k` and uncompressed WIFs start with the character `K`.
- privkey_from_wif takes a Qeditas WIF string and returns the corresponding private key (as a big integer) along with a boolean indicating if it is for the compressed address.
- privkey_from_btcwif takes a Bitcoin WIF string and returns the corresponding private key (as a big integer) along with a boolean indicating if it is for the compressed address. This function is included to make it easy for people to import Bitcoin private keys corresponding to the initial distribution.
- pubkey_hashval takes a non-zero public key (a pair (x, y)) and a boolean (indicating if the compressed address should be used) and returns the 20 bytes which result from hashing either the compressed public key (2 with x if y is even; 3 with x if y is odd) or the uncompressed public key (4 with x and y). This hash value is also the corresponding pay to public key hash address.
- hashval_from_addrstr takes a string with a Qeditas address and returns the underlying hash value.
- hashval_btcaddrstr takes a hash value and returns the corresponding Bitcoin address.

- addr_qedaddrstr takes a Qeditas address and returns a base 58 string representation of the address. The prefix byte used is different for the four different kinds of addresses:
  - Pay to public key hash addresses use a prefix byte of 58 and so these Qeditas addresses begin with the character `Q`.
  - Pay to script hash addresses use a prefix byte of 120 and so these Qeditas addresses begin with the character `q`.
  - Term addresses use a prefix byte of 66 and so these Qeditas addresses begin with the character `T`.
  - Publication addresses use a prefix byte of 56 and so these Qeditas addresses begin with the character `P`.
- qedaddrstr_addr takes a string with a base 58 Qeditas address and returns the Qeditas address.
- btcaddrstr_addr takes a string with a base 58 Bitcoin address (either p2pkh or p2sh) and returns the Qeditas address.

Some of the code in this module was taken from the code for Egal [3]. Egal included `BIP 32` code that isn't needed in Qeditas. Egal relied on openssl to compute `SHA256` and `ripemd160` hashes, but Qeditas does this itself.

6.3 Cryptographic Signature Checking

The module signat implements functions for creating and verifying cryptographic signatures over the elliptic curve. A cryptographic signature (represented by the type signat) is a pair (r, s) of big integers. As usual, the functions seo_signat and sei_signat serialize and deserialize elements of type signat. Let n be the order of the group for `secp256k1`.

- decode_signature takes a base 64 string and returns $(i, c, (r, s))$ where $i \in \{0, 1, 2, 3\}$ ("recid") is a tag to help recover the public key from the signature and what was signed, c ("fcomp") is a boolean indicating if the signature is for a compressed public key and (r, s) is the cryptographic signature.
- signat_big_int takes a big integer $e < n$ (in practice $e < 2^{160}$), a big integer private key $k < n$ and a random big integer $R < n$ and returns a signature (r, s). The signature (r, s) signs e with the private key k.
- signat_hashval is the same as signat_big_int except it is given a hash value h to sign instead of a big integer. The implementation simply converts h to a (160-bit) big integer using hashval_big_int and calls signat_big_int. The result is a signature (r, s) signing h with the given private key.

- verify_signed_big_int takes a big integer e, a point (public key) (x, y) and a signature (r, s) and returns a boolean indicating if (r, s) is a valid signature of e by the private key corresponding to (x, y).

- verify_p2pkhaddr_signat takes a big integer e, a p2pkhaddr α (equivalently, a hash value), a signature (r, s), an integer $i \in \{0, 1, 2, 3\}$ and a boolean c. It uses e, (r, s) and i to recover a point on the curve using recover_key. If recover_key returns the zero point, then the signature is not valid and the boolean false is returned. Otherwise, recover_key returns a public key (x, y). The p2pkhaddr corresponding to (x, y) (compressed if c is true, uncompressed otherwise) is computed using pubkey_hashval and compared with α. If they are the same, then the signature is valid and the boolean true is returned. Otherwise, the signature is not valid and the boolean false is returned.

- verifybitcoinmessage is used to verify a bitcoin signed message returning a boolean (true if valid, false otherwise). The inputs are a p2pkhaddr α, $i \in \{0, 1, 2, 3\}$, a boolean c, a cryptographic signature (r, s) and a string m (the message). If Qeditas is running in the testnet, then the message is prefixed with `testnet:`. This allows people to sign, for example, endorsements which are valid on the testnet, but not on the mainnet. The message is then modified the same way as the Bitcoin core client (essentially including `Bitcoin Signed Message:` and some characters indicating the length of this prefix and the length of the message). The remainder of the work is performed by the internal verifymessage function: The message is double `SHA256` hashed and converted to a big integer e. The public key is attempted to be recovered using e, (r, s) and i using recover_key (with false returned upon failure). Assuming the public key (x, y) is recovered, the final check verifies that the hash of the public key (compressed if c is true, uncompressed otherwise) is α.

- verifybitcoinmessage_recover is used to verify a bitcoin signed message returning an optional public key (x, y) (the corresponding public key if the signature is valid, the `None` if not). It behaves equivalently to verifybitcoinmessage except upon success it returns the public key as `Some`(x, y) and returns `None` upon failure. In this case, internal verifymessage_recover function is used.

There is also an internal function recover_key which computes a public key (x, y) from a big integer e (from the hash value of what was signed), a signature (r, s) and a "recid" $i \in \{0, 1, 2, 3\}$. This should be the public key corresponding to the private key which was used to construct (r, s) from e.

Note: The Coq module CryptoSignatures is intended to correspond to the signat module. It defines a Coq type signat and functions to simulate signing with a private key and checking a signature. The actual implementation is trivial, but only the required properties are exported.

6.4 Scripts and Generalized Signatures

The module script implements the Bitcoin scripting language (with the exceptions of OP_SHA1 and OP_RIPEMD160). The main reason the Bitcoin scripting language is included is so p2sh addresses in the Bitcoin snapshot can be redeemed. Scripts are represented by lists of integers which should be bytes. The following functions operate on scripts:

- hash160_bytelist takes a script and computes its hash by taking the SHA256 and then RIPEMD160. The hash value returned should be interpreted as a p2shaddr. The procedure is the same way Bitcoin computes p2sh addresses.

- eval_script evaluates a script in context. The inputs are a big integer e (which corresponds to what is meant to be "signed"), the script $\overline{s}$ and two stacks. The function returns the two stacks which result from evaluating the script.

- verify_p2sh compares a script's hash against the given p2shaddr and verifies that the script evaluates to "true." A boolean is returned. The inputs are a big integer e (which corresponds to what is meant to be "signed"), a p2shaddr β and a script $\overline{s}$. The only way true will be returned is if the following occurs:

  1. The script is evaluated and the top of the main stack is a script $\overline{s_1}$ which hashes to give β.
  2. The script $\overline{s_1}$ is popped off the main stack and evaluated leaving something nonzero at the top of the main stack.

The process of evaluating the script is more complicated than it is in Bitcoin. The reason is that `OP_CHECKSIG` and `OP_CHECKMULTISIG` may be signatures by endorsement. An endorsement may be, for example, an endorsement of a p2sh address to a p2pkh address. In order to check the endorsement, another script must be checked to be a valid "signature" of a different value (the hash of the endorsement message). For this reason, the main functions that do the work: eval_script, eval_script_if, checksig, checkmultisig and check_p2sh are mutually recursive.

In order to account for endorsements in a uniform way, the type gensignat of "generalized signatures" is defined. There are six constructors of type gensignat corresponding to six ways of making a signature:

- P2pkhSignat is an ordinary cryptographic signature (r, s) corresponding to a given public key (x, y) which may or may not be compressed. (Note that the public key is explicitly given here and need not be recovered during signature checking.) The public key corresponds to a `p2pkhaddr` (again, one compressed and one uncompressed).

- P2shSignat is a script and should be checked by calling verify_p2sh above. The script corresponds to a certain `p2shaddr`. (Note that the correspondence is not direct. The script itself is not hashed to obtain the `p2shaddr`. Instead the script should evaluate to yield a script which hashes to the `p2shaddr` at the top of the main stack.)

- EndP2pkhToP2pkhSignat This is a p2pkh signature via a p2pkh endorsement. This means that two public keys (and two booleans indicating compressed or uncompressed) are given along with two signatures. One of the signatures is of a Bitcoin message with the appropriate base 58 Qeditas pay to public key hash address, signed with the private key for the other public key. The other signature is the signature of what should be signed.

- EndP2pkhToP2shSignat This is a p2sh signature via an endorsement a p2pkh endorsement. That is, a signature of a Bitcoin message with the appropriate base 58 Qeditas pay to script hash address is given, corresponding to the public key of the p2pkh address. Also, a script corresponding to the p2sh address is given which "signs" what should be signed.

- EndP2shToP2pkhSignat This is a p2pkh signature via an endorsement a p2sh endorsement. Here a script is given which "signs" the Bitcoin message with an endorsement to a base 58 Qeditas pay to public key hash address. An ordinary cryptographic signature (corresponding to the p2pkh address) signing what is to be "signed" is given.

- EndP2shToP2shSignat This is a p2sh signature via an endorsement a p2sh endorsement. Here a script is given which "signs" the Bitcoin message with an endorsement to a base 58 Qeditas pay to script hash address. A separate script corresponding to the other p2sh address is given which "signs" what should be signed.

As usual, there are two functions seo_gensignat and sei_gensignat for serializing generalized signatures.

There is one important function for working with generalized signatures:

- verify_gensignat takes a big integer e (corresponding to the hash value of what should be signed), a generalized signature and an address, and returns a boolean indicating if the generalized signature verifies that the owner of the address "signed" e. The address should be either a pay to public key hash address or pay to script hash address. (If a term address or publication address is given, false is returned.) Each of the six cases is considered separately in the code, but the idea is clear: check that the signature corresponds to the given address and check that the signature "signs" e.

Chapter 7

Networking

The intention of the net module (`net.ml` and `net.mli`) is to create and handle connections to remote nodes. Not everything has been implemented, but there is enough implemented to form connections between peers and pass messages containing inventory, transactions, block headers and block deltas.

7.1 Network Messages

There are several different message types enumerated by the type msgtype. These largely follow the message types of Bitcoin's messages.

- Version: used (only) in the handshake protocol, which is similar to Bitcoin's handshake.
- Verack: used (only) in the handshake protocol, which is similar to Bitcoin's handshake.
- Addr: intended to be used as in Bitcoin, but is currently unused.
- Inv: inventory message with a list of triples (m, h, k) where m is a message type (as a byte), h is a block height (which is often irrelevant) and k is a hash value (identifying some corresponding data).
- GetData: currently unused
- MNotFound: currently unused
- GetSTx: request a transaction with its signatures (see Chapter 10).
- GetTx: request a transaction (see Chapter 10).
- GetTxSignatures: request signatures for a transaction (see Chapter 10).
- GetHeader: request a block header (see Chapter 12).

- GetHeaders: request several block headers (see Chapter 12).
- GetBlock: request a block (currently unused, as headers and deltas are requested independently).
- GetBlockdelta: request a block delta (see Chapter 12).
- GetBlockdeltah (deprecated)
- STx: a transaction with its signatures (see Chapter 10).
- Tx: a transaction (see Chapter 10).
- TxSignatures: signatures for a transaction (see Chapter 10).
- Block: a block (currently unused, as headers and deltas are sent independently).
- Headers: a block header (see Chapter 12).
- Blockdelta: a block delta (see Chapter 12).
- Blockdeltah (deprecated)
- GetAddr: currently unused
- Mempool: currently unused
- Alert: currently unused
- Ping: intended to be used to ensure a connection is alive, but currently unused.
- Pong: intended to be used to ensure a connection is alive, but currently unused.
- Reject: currently unused
- GetCTreeElement: request an compact tree element (see Chapter 11).
- GetHConsElement: request an hcons element (see Chapter 11).
- GetAsset: request an asset (see Chapter 10).
- CTreeElement: an compact tree element (see Chapter 11).
- HConsElement: an hcons element (see Chapter 11).
- Asset: an asset (see Chapter 10).
- Checkpoint: a checkpoint (see Chapter 13).
- AntiCheckpoint: currently unused
- NewHeader: sending a newly staked header (deprecated)

- GetCheckpoint: request a checkpoint (see Chapter 13).

Message types can be converted to integers and strings by int_of_msgtype and string_of_msgtype, respectively. msgtype_of_int computes the message type of an integer (raising Not_found if the integer is negative or greater than 36).

The function inv_of_msgtype computes int_of_msgtype for a converted message type. For example, GetTx is converted to Tx. In general the message type is converted from a request message type to a response message type in order to keep up with what inventory a remote peer has advertised in relation to what inventory has been requested.

Messages have the following format:

- Four byte magic number: 0x51656454 for testnet, 0x5165644d for mainnet.
- Either the byte 0 or the byte 1 followed by a 20 byte hash value h indicating that the message is a reply to a specific request (identified by h).
- A byte indicating the message type (msgtype).
- Four bytes giving the length l of the payload.
- 20 bytes giving the hash of the payload.
- The payload in l bytes, which must hash (via hash160) to give the correct hash value.

Three exceptions are exported from net:

- GettingRemoteData: indicates some data is being requested from peers and this data is required in order to complete the current computation. This exception is not used in net itself, but is raised by other code when a request is sent to a peer and the computation will not wait for a reply.
- RequestRejected: indicates that a socks4 connection failed.
- IllformedMsg: indicates a message had a bad format.

The internal function rec_msg reads a message from a channel, given a block height. The block height is required to know the maximum allowed length of a message payload (computed from maxblockdeltasize). Since the current block height changes, there is an exposed reference netblkh which is used by connection threads to determine the current block height. The value netblkh is updated by code that tracks the current best block (see Chapters 13 and 15). The exception IllformedMsg is raised by rec_msg unless the message has the correct format.

The internal function send_msg handles formatting and sending a message. In particular, it is given an output channel, the hash of the payload, an optional hash indicating what the message is a reply to, the message type and the message payload as a string (to be interpreted as a sequence of bytes).

7.2 Network Connections

Network connections have a state summarized in the record type connstate consisting of the following fields:

- conntime: the time the connection was created.
- realaddr: the address of the connecting socket.
- connmutex: a "mutex" to prevent race conditions when communicating with the peer.
- sendqueue: messages on the queue to be sent by the connection sender thread to the peer.
- sendqueuenonempty: a condition to wake up the connection sender thread to send a message on the queue.
- handshakestep: an integer indicating the current handshake step, which progresses to 5 at which point the handshake protocol has been completed.
- peertimeskew: the number of seconds of skew between the peers (computed from the advertised time in the handshake).
- protvers: the protocol version advertised by the peer in the handshake.
- useragent: the user agent given by the peer in the handshake.
- addrfrom: the string giving the address the peer gave in the handshake.
- banned: a boolean which becomes true if the connection is banned due to misbehavior.
- lastmsgtm: the time of the last message exchanged.
- pending: a list of message hashes where the peer is expected to give a response, as well as information about how long the node has waited and a function for handling the reply if and when it is received.
- sentinv: a list of inventory already sent to the peer
- rinv: a list of remote inventory
- invreq: a list of requested inventory
- first_header_height: intended to be the block height at which the peer has headers (currently unused)
- first_full_height: intended to be the block height at which the peer has block headers and deltas (currently unused)
- last_height: indended to be the latest block height of the peer (currently unused)

All current connections are stored in the ref list netconns. Each entry of netconns has two threads (the "connection listener" thread and the "connection sender" thread), a file descriptor and its corresponding input and output channels, and a reference to an optional connection state (connstate). If the reference becomes None, then the connection is considered dead.

7.3 Network Listener

If an ip address to listen for connections is given, the function openlistener creates a listener socket waiting for connections. This listener socket is sent to the function netlistener in its own thread upon startup (see `qeditas.ml`). The variable netlistenerth contains the thread in case it is required later.

The function netlistener is an infinite loop accepting connections and removing dead connections (those with no connection state) using remove_dead_conns. A connection is accepted if it is not a self-connect (as judged by the value of this_nodes_nonce) and if there are fewer than maxconns connections on netconns. An initial connection state (connstate) is created with handshakestep set to 1. A connection listener thread is created calling connlistener and a connection sender thread is created calling connsender. The triple of the two threads and (a reference to the optional) connection state is added to netconns.

connsender is a loop which sends the messages on sendqueue to the peer and then waits for the condition sendqueuenonempty. If at any time the connection state of the connection is set to None, the thread ends. Likewise, certain exceptions (Unix_error, End_of_file, ProtocolViolation, SelfConnection) result in closing the socket, setting the connection state to None and ending the thread. All other exceptions are logged but otherwise ignored.

connlistener is a loop which waits for messages (using rec_msg) and calls handle_msg to handle the message. If after handling the message banned is true, then the exception ProtocolViolation is raised. If at any time the connection state of the connection is set to None, then the connection is considered dead and End_of_file is raised. Certain exceptions (Unix_error, End_of_file, ProtocolViolation, SelfConnection) result in closing the socket, setting the connection state to None and ending the thread. All other exceptions are logged but otherwise ignored.

The function handle_msg behaves differently depending on whether the message is a reply. If the message is a reply to a message with hash value h, then the associated value is looked up on the pending field of the connection state (and removed from pending). This gives a function f which is called with the message type and message payload. If no associated value is on pending, the message is ignored.

If the message is not a reply, then a distinction is made based on whether or not the connection is still in the handshake phase. If the connection is in the handshake phase, then the message must be either a Version or Verack message and is handled appropriately, updating the connection state. Once the handshake protocol has been completed, handshakestep will be 5 and the

function given by send_inv_fn is called to send the initial inventory message to the peer. (Initially, send_inv_fn is set to a trivial function that does nothing, but the value is reset to the send_inv function in blocktree. send_inv in blocktree gathers recent block headers, block deltas and transactions to send as inventory.)

If the connection is not in the handshake phase, then a function f to handle the message type is looked up in the hash table msgtype_handler and this f is called on the channels, the connection state and the message payload. If there is no handler function is found in msgtype_handler, then Failure is raised.

The particular functions associated with message types in msgtype_handler are given in later modules (blocktree, ctre and assets). (The reason is that only in later modules are the types for the deserialized data in the message available.) In many cases, no handler is given (due to the incomplete nature of the Qeditas implementation) meaning the messages will lead to a Failure exception being raised.

7.4 Network Seeker

The function netseeker is used to start a thread which tries to use known peers (from past connections stored in a `peers` file) or fallback nodes (see testnetfallbacknodes and fallbacknodes) to initiate new connections. The main code is in netseeker_loop. The thread is stored in netseekerth in case it is needed later.

7.5 Other Exported Network Functions and Data

There are a few important functions called by other modules in order to communicate with peers.

- peeraddr: return the address of an optional given connstate as a string.
- tryconnectpeer: given an address of a peer, try to connect to the peer, adding it to netconns if successful. Upon success, return the triple added to netconns. Upon failure, return None.
- addknownpeer: update the hash table knownpeers to either add the new peer or to update the latest connection time of an already known peer.
- removeknownpeer: remove a given peer from the knownpeers hash table.
- getknownpeers: iterated over the knownpeers hash table and collect the first 1000 peers which have been active in the past week.
- loadknownpeers: load the `peers` file (if it exists) and add the peers which have been active in the past week to the knownpeers hash table.
- saveknownpeers: create a new `peers` file consisting of the contents of the knownpeers hash table (as a text file with the peer's address followed by the last time the connection was active).

- BannedPeer is an exception which is raised when trying to connect to a banned peer.
- bannedpeers is a hash table remembering the addresses of banned peers to prevent reconnection.
- banpeer adds an address to bannedpeers.
- clearbanned removes all addresses from bannedpeers.
- network_time: returns the current local time modified by the median skew time of all connected nodes.
- queue_msg: puts a non-reply message onto the sendqueue of a connection.
- queue_reply: puts a reply message onto the sendqueue of a connection.
- find_and_send_requestdata: searches through the current peers to find one that has some desired data in its inventory (but has not already had the data requested). If one is found, the data is requested. Otherwise, Not_found is raised.
- broadcast_requestdata: request some data from all peers that have the data in the inventory and from whom the data has not already been requested.
- broadcast_inv: send an Inv (inventory) message to all peers.

Chapter 8

Database

For several data types we will need to manipulate persistent storage of values indexed by a hash value. (We will call this a "database" although it is only a key-value mapping.) One way to do this would be to use a standard library built for this purpose, such as leveldb. However, integrating leveldb with the OCaml code has proven challenging. Instead (at least for the moment) the database has been implemented by simply using files in directories. The particular implementation has been abstracted using a module type (dbtype) so that the implementation of the module can be easily replaced. The first implementation, Dbbasic, was due to Trent Russell in early 2016. Since Dbbasic requires many file operations (to keep index files sorted), a modified version Dbbasic2 was added by Bill White later in 2016. Dbbasic2 requires significantly more RAM as index files are loaded into RAM upon initialization,[1] but can simply append new keys to index files since they do not need to remain sorted in the file. Both implementations are still included in the db module and each database can use whichever implementation seems more appropriate. At the moment, all use Dbbasic except the database for headers (since this requires an additional function to iterate over keys).

The module type dbtype is actually a functor type. It depends on a signature with

- a type `t` (the type of the values to be stored),
- a string `basedir` (indicating the top level directory where these key-value pairs will be stored) and
- functions `seival` and `seoval` for deserializating and serializing the data from and to channels.

An implementation of dbtype must implement the following:

- dbinit is intended to be called once, upon startup. It searches for all `index` and `deleted` files of the database and loads the contents into hash tables.

[1] The initial load of the indexing data is handled by a function dbinit called at startup, which was not in the first version of dbtype.

- dbget taking a hash value (as the key) to a value of type `t` (or raising Not_found).
- dbexists takes a hash value (as the key) and returns true if there is an entry with this key and returns false otherwise. (One could use dbget for this purpose, but dbget must take the time to deserialize corresponding the value.)
- dbput takes a hash value (as the key) and a value of type `t` and stores the key-value pair.
- dbdelete takes a hash value (as the key) and deletes the entry with this key, if one exists. If there is no entry, dbdelete does nothing.

In addition there is a module type dbtypekeyiter which requires the implementation of an extra function dbkeyiter. The function dbkeyiter applies a function to every key in the database. This is used in practice to initialize headers on startup.

The two functors Dbbasic and Dbbasic2 return a module implementing dbtype, given an implementation of `t`, `basedir`, `seival` and `seoval`. The implementation of both Dbbasic and Dbbasic2 use subdirectories of `basedir` with three files: `index`, `data` and `deleted`. The file `data` contains serializations of the values stored in this directory and the file `index` contains the keys (hash values) along with integers giving the position of the corresponding data in `data`. The file `deleted` is a list of heys (hash values) that have been marked as deleted (but the keys are still in `index` and the value is still in `data`).

The maximum number of entries in the files in a directory is 65536, but new entries are also not allowed after the `data` file exceeds 100 MB. After no more entries are permitted in a directory, a subdirectory named using the next byte (in hex) of the key is created (if necessary) and this subdirectory is used, unless it is also full.

Some auxiliary functions are used:

- find_in_deleted checks if a key is in the `deleted` file of a directory.
- find_in_index searches for a key by loading the `index` file and doing a binary search. If it is found, then the position of the value in the `data` file is returned. Otherwise, Not_found is raised.
- load_index loads the index file as a list of pairs of hash values and positions. This is only used by Dbbasic (not by Dbbasic2) and assumes the index file is sorted, in which case it returns the list reverse sorted by the hash values.
- load_index_to_hashtable takes a hash table and a directory d. Assuming an `index` file is in the directory, each entry is inserted into the hash table. That is, for each key k and integer p (giving the position of the data in the `data` file in the directory), the hash table associates the key k with the directory d and position p.

- count_index gives the number of entries in the `index` file of a directory.
- load_deleted_to_hashtable takes a hash table and a directory. Assuming a `deleted` file is in the directory, each key k in the file is added to the hash table.
- load_deleted loads all the hash values (keys) in the `deleted` file of a directory.
- undelete removes a key from the `deleted` file of a directory by loading all the deleted keys and then recreating the `deleted` file without given the key.
- count_deleted gives the number of entries in the `deleted` file of a directory.
- dbfind and dbfind_a are used to search for a subdirectory where the index file contains a given key, returning the directory and value position if one is found. If none is found, Not_found is raised.
- dbfind_next_space, dbfind_next_space_a and dbfind_next_space_b are used to find the next appropriate subdirectory and position where a key can be included.
- defrag cleans up by actually deleting key-value pairs which have been deleted. (This is only called by Dbbasic.)

The implementation of Dbbasic works as follows:

- There are two hash table cache1 and cache2 which store (roughly) the last 128 to 256 entries looked up so that these can be returned again quickly. (When one cache has 128 entries, the other is cleared and begins to be filled.) The internal functions add_to_cache and del_from_cache handle adding and removing key-value pairs from the cache.
- dbget tries to find the key in the cache. If it is not in the cache, dbfind is called to try to find a subdirectory and value position. If one is found and the key has not been deleted, then the value is deserialized from the `data` file starting at the position and returned. Otherwise, Not_found is raised.
- dbexists is analogous to dbget except it does not deserialize the value if it is found. Instead it returns true if a subdirectory and value position were found (and the key is not marked as deleted), and returns false otherwise.
- dbput takes a key k and value v. The function dbfind is called to find an entry for k if one exists. If one is found and it has been marked as deleted, then undelete it. If one is found and it has not been deleted, then simply return – as the key value pair already exists. Otherwise, call dbfind_next_space to find the next subdirectory and position where the new value can be stored (which is at the end of the `data` file, or 0 if no `data` file yet exists). The function load_index at this subdirectory is used to get

a reverse sorted list of the current keys and positions. The list is reversed and the new key and position are merged into the list. This new list is stored in `index` replacing the previous contents. The value is deserialized and appended to the end of the `data` file.

- dbdelete takes a key and uses dbfind to find a subdirectory where the corresponding `index` file contains the key. If none is found, then do nothing. Assume a subdirectory is found. If the key is already in the `deleted` file of this subdirectory, then do nothing. Otherwise, append the key to the `deleted` file. If the number of deleted entries in this subdirectory exceeds 1024, then defrag is called.

The implementation of Dbbasic2 works as follows:

- A "mutex" mutexdb is created and used with a function withlock in a way that is intended to make the code thread-safe. In particular, this should be used when reading from or writing to database files.

- A hash table indextable is created to associate keys k with a pair (d, p) where d is a string giving a directory which should contain a `data` file. The value associated with k should be contained at position p in the `data` file.

- A hash table deletedtable is created to remember which keys have been deleted. (The corresponding values are not deleted from the `data` file or `index` file. In principle they could be deleted by cleanup code offline.)

- The functions dbinit and dbinit_a traverse the directories under the main database directory calling loading the contents of the `index` files into indextable and the contents of the `deleted` files into deletedtable.

- dbexists checks if a given key is in indextable and not in deletedtable.

- dbget finds (d, p) associated with k in indextable. If no (d, p) is found, Not_found is raised. If k is in deletedtable, Not_found is raised. Otherwise, the data starting at byte p in the `data` file in the directory d is deserialized and returned.

- dbput takes a key k and value v. Suppose some (d, p) is associated with k in indextable. If k is in deletedtable, then it is removed from deletedtable and the corresponding entry is removed from the `deleted` file (using undelete). If k is not in deletedtable, then simply return – as the key value pair already exists. If no (d, p) is associated with k in indextable, call dbfind_next_space to find the next subdirectory and position where the new value can be stored (which is at the end of the `data` file, or 0 if no `data` file yet exists). The new entry (k, p) is appended to the `index` file and the pair (d, p) is associated with k in the indextable hash table. The value is deserialized and appended to the end of the `data` file.

- dbdelete takes a key k and checks if a corresponding (d, p) is in indextable. If no entry is found, then do nothing. Assume some (d, p) is found. If k is in deletedtable, then do nothing (as it has already been deleted). Otherwise, add k to deletedtable and append k to the `deleted` file of the directory d.

There is also a module Dbbasic2keyiter of module type dbtypekeyiter implemented the same way as Dbbasic2 except with the additional function dbkeyiter which takes a function f and calls it on every key in the hash table indextable unless the key is in deletedtable.

Two simple database modules DbBlacklist and DbArchived are defined by giving Dbbasic the type bool and base directories `blacklist` and `archived`, respectively. DbBlacklist is intended to save keys corresponding to some blacklisted data that should not be requested from peers. DbArchived is intended to save keys corresponding to old data the node no longer wishes to store or receive.

Other instances of Dbbasic occur where the corresponding data types are defined. For assets this is in assets and for (signed) transactions this is in tx (see Chapter 10). For hcons elements and ctree elements, giving approximations of parts of ledger trees, this is in ctre (see Chapter 11). For block deltas, this is in block (see Chapter 12). Block headers use Dbbasic2keyiter instead of Dbbasic so that hashes of all block headers can be processed during initialization. The corresponding database DbBlockHeader is defined in block (see Chapter 12).

Users can change each use of Dbbasic to be Dbbasic2, as desired, and Qeditas should still work. Note, however, that trying to change back from Dbbasic2 to Dbbasic may not work, as the index files in the relevant database may no longer be sorted.

Chapter 9

Formalized Mathematics

The mathdata module contains the code for representing types, terms and proofs and the code for type checking and proof checking. This is arguably the most important module in Qeditas. A bug in this module could lead to non-theorems being accepted as theorems undermining the primary purpose of the Qeditas system. Fortunately the `mathdata.ml` file is not long (currently less than 1500 lines of code) and depends very little on the rest of Qeditas (using only code for serialization and cryptographic hashing). It is intended to satisfy the *de Bruijn criterion* in that the code can be manually audited to determine its correctness. We attempt to give enough information in this chapter for someone who wishes to undertake such an audit.

The original version of the code for this module was taken from the code for the Egal system [3], but has since undergone extensive changes. One major difference in the syntax is the explicit support for type variables in Qeditas. Support for theories and signatures have also been added, and the type of documents has been modified (adding support for importing signatures and declaring conjectures but removing all presentation level items). Additionally, the checking functions are parameterized by functions to verify a term identified only by its hash root has a type in a theory and to verify a proposition identified only by its hash root is known to be a theorem in a theory. Such information will be looked up in the ledger tree (see Chapter 11) by checking what is held at corresponding term addresses. Finally, a significant portion of the Egal proof checking code was apparently intended to avoid expanding definitions unnecessarily. This code has been deleted and replaced by simpler code to expand all definitions during proof checking.

One might argue that it would be safer to use an older, established proof checker. However, experience has shown that even established systems can be vulnerable to "tricks" which can be used to prove what should be a non-theorem. For example, on `proofmarket.org` [29] a bitcoin bounty was placed on the proposition False in Coq [18]. In spite of the fact that Coq is an advanced

tool used by many people for many projects, such a "proof" of False was given.[1] The "proofs" were related to implementation issues rather than an inconsistency in the underlying logic, but only the implementation will matter in a system like Qeditas. By using a simple underlying logic (simple type theory) and isolating the implementation in the reasonably small module mathdata it is hoped that such apparent inconsistencies can be avoided.

The underlying logic is a form of simple type theory [5] with support for prefix polymorphism. The basic proof calculus is natural deduction [9, 25] with Curry-Howard style λ-terms proof terms [17]. This leads the type checker and proof checker in mathdata to be very similar to the oldest proof checker, AUTOMATH [7]. The logic is designed to allow for multiple theories to be declared and for signatures to be used to import previous typed terms and proven propositions. Of the popular proof assistants at the moment, the closest would probably be Isabelle [22], although Isabelle follows the LCF style [13] instead of Curry-Howard.

Note: Unit tests for the mathdata module are in `mathunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches. A few examples of types, terms and proof terms used in these unit tests are in `unittestsaux.ml` in the same directory. Likewise, examples of publications (encoded versions of documents released with Egal [3]) are in `testpubs1.ml` and `testpubs2.ml` in the same directory.

Note: The Coq module MathData is intended to correspond to mathdata, except that the checking code is omitted and left abstract.

9.1 Simple Types

Simple types (α, β) are described by the following grammar:

$$\alpha, \beta ::= \delta_n | o | \iota_n | (\alpha \to \beta) | (\Pi \alpha)$$

We treat $\to$ as right associative to omit parentheses. For example, $\iota_0 \to \iota_0 \to o$ means $(\iota_0 \to (\iota_0 \to o))$. Also, we will omit parentheses in $\Pi\alpha$ since Π will always be used above $\to$ and so no ambiguity can result.

Simple types are implemented as the inductive type tp. We describe each constructor:

- TpVar(n) means the type variable δ_n, where the n should be interpreted as a de Bruijn index [4]. For example, $\Pi\Pi\delta_1 \to \delta_0 \to \delta_1$ means the type of a function which expects two types α and β, a term of type α, a term of type β and returns a term of type α.

- Prop means the type o of propositions.

[1] In fact, two different proofs were given.

- Base(n) means the n^{th} base type ι_n. Only finitely many base types will be explicitly used in a theory. In fact, so far only theories using one base type ι_0 have been considered, but the support for multiple base types is included in case it is needed later.

- TpAll(α) means $\Pi\alpha$, binding a type variable. Only types of the form $\Pi \cdots \Pi\alpha$ where α has no occurrence of a Π will be used in practice.

The functions seo_tp and sei_tp serialize and deserialize types.

hashtp takes a type and returns a hash value obtained by serializing the type to a string, hashing the string, and then hashing the result tagged with 64. (The intention of hashing tagged results is to ensure that, for example, the hash value associated with a type will not accidentally be the same as the the hash value associated with a term, proof or anything else.)

9.2 Terms and Propositions

Terms s, t, u are described by the following grammar:

$$s, t ::= x_n | \sharp_h | c_n | (st) | (\lambda_\alpha s) | (s \to t) | (\forall_\alpha s) | (s\alpha) | (\Lambda s) | (\hat{\forall} s)$$

Here n ranges over non-negative integers and h ranges over hash values.

Terms x_n are variables, where n should be interpreted as a de Bruijn index [4]. For example, $\lambda_o x_0 \to \forall_o x_1 \to x_0$ would be written as $\lambda y : o.y \to \forall z : o.y \to z$ in a named representation. A term $\sharp_h$ is an abbreviation for a term which has h as its hash root (see tm_hashroot below). Note that there are two kinds of application: (1) (st) of a term s to a term t and (2) $(s\alpha)$ of a term s to a type α. Likewise there are two abstractions and two universal quantifiers: one for the term level and one for the type level. First, $(\lambda_\alpha s)$ is a term level abstraction representing a function expecting an input of type α with return value determined by this input and s. Likewise, $(\forall_\alpha s)$ corresponds to universally quantifying over the elements of type α. On the other hand, (Λs) is a type level abstraction and represents a function which expects a type α and then returns a value determined by α and s. Likewise, $(\hat{\forall} s)$ corresponds to universally quantifying over all types. We refer to λ_α, $\forall_\alpha$, Λ or $\hat{\forall}$ collectively as *binders* and say the term s in $(\lambda_\alpha s)$, $(\forall_\alpha s)$, (Λs) or $(\hat{\forall} s)$ is in the *scope* of the binder.

We often omit parentheses. Application is assumed to be left associative and so $s\alpha\beta tu$ means $((((s\alpha)\beta)t)u)$ If parenthesis around the body of a binder are omitted, then they are assumed to be such that the scope of the binder is as large as possible. For example, $\forall_o x_0 \to x_0$ means $(\forall_o (x_0 \to x_0))$.

The corresponding type in the OCaml code is tm. We describe each constructor:

- DB(n) corresponds to the variable x_n (i.e., the de Bruijn index).

- $\mathsf{TmH}(h)$ corresponds to the term $\sharp_h$ and should be considered an abbreviation (which is sometimes opaque and sometimes transparent, depending on the current signature).
- $\mathsf{Prim}(n)$ corresponds to the primitive c_n.
- $\mathsf{Ap}(s,t)$ corresponds to term level application st.
- $\mathsf{Lam}(\alpha,s)$ corresponds to term level abstraction $\lambda_\alpha s$.
- $\mathsf{Imp}(s,t)$ corresponds to implication $s \to t$.
- $\mathsf{All}(\alpha,s)$ corresponds to term level universal quantification $\forall_\alpha s$.
- $\mathsf{TTpAp}(s,\alpha)$ corresponds to type level application $s\alpha$.
- $\mathsf{TTpLam}(s)$ corresponds to type level abstraction Λs
- $\mathsf{TTpAll}(s)$ corresponds to type level universal quantification $\hat{\forall} s$.

The functions seo_tm and sei_tm serialize and deserialize terms.

There are two functions hashtm and tm_hashroot which take terms and return a corresponding hash value. In the case of hashtm, a hash value is obtained by serializing the term to a string, hashing the string, and then hashing the result tagged with 66. This (effectively) guarantees that different terms will always be given different hash values. On the other hand, tm_hashroot takes a term and computes its *hash root*. The hash root of a term does not distinguish between a term $\sharp_h$ and a term t which has h as its hash root. In effect, tm_hashroot views all such abbreviations as transparent.

The *hash root* $\mathbf{TR}_\sharp(t)$ of a term t can be defined as follows:

- $\mathbf{TR}_\sharp(\sharp_h)$ is h
- $\mathbf{TR}_\sharp(c_n)$ is the hash of n tagged with 96.
- $\mathbf{TR}_\sharp(x_n)$ is the hash of n tagged with 97.
- $\mathbf{TR}_\sharp(st)$ is the hash of the hashed pair of $\mathbf{TR}_\sharp(s)$ and $\mathbf{TR}_\sharp(t)$ tagged with 98.
- $\mathbf{TR}_\sharp(\lambda_\alpha s)$ is the hash of the hashed pair of the hash of α and $\mathbf{TR}_\sharp(s)$ tagged with 99.
- $\mathbf{TR}_\sharp(s \to t)$ is the hash of the hashed pair of $\mathbf{TR}_\sharp(s)$ and $\mathbf{TR}_\sharp(t)$ tagged with 100.
- $\mathbf{TR}_\sharp(\forall_\alpha s)$ is the hash of the hashed pair of the hash of α and $\mathbf{TR}_\sharp(s)$ tagged with 101.
- $\mathbf{TR}_\sharp(s\alpha)$ is the hash of the hashed pair of $\mathbf{TR}_\sharp(s)$ and the hash of α tagged with 102.

- $\mathbf{TR}_\sharp(\Lambda s)$ is the hash of $\mathbf{TR}_\sharp(s)$ tagged with 103.
- $\mathbf{TR}_\sharp(\hat{\forall} s)$ is the hash of $\mathbf{TR}_\sharp(s)$ tagged with 104.

The reader can verify that this corresponds to the definition of tm_hashroot in the code. The tags are used to record which term constructor was traversed and is also used to ensure that hash roots of terms are not the hash values computed in other contexts.

A proposition is a certain kind of term (in a given context). In short, propositions are always of the form $\hat{\forall} \cdots \hat{\forall} t$ where t has type o. Usually a proposition is simply of the form t where t has type o.

9.3 Proof Terms

Proof terms $\mathcal{D}, \mathcal{E}$ are described by the following grammar:

$$\mathcal{D}, \mathcal{E} ::= \hat{\sharp}_h | \mathbf{H}_n | \Box_h | (\mathcal{D}s) | (\mathcal{D}\mathcal{E}) | (\lambda_s \mathcal{D}) | (\lambda_\alpha \mathcal{D}) | (\mathcal{D}\alpha) | (\Lambda \mathcal{D})$$

Here n ranges over non-negative integers and h ranges over hash values. We sometimes simply say "proofs" instead of "proof terms."

The proof term $\hat{\sharp}_h$ is an abbreviation for a proof term which has hash root h (see pf_hashroot below). The proof term $\mathbf{H}_n$ is the proof of a hypothesis (in a hypothesis context). The proof term $\Box_h$ simply asserts that the proposition with hash root h is known. (The current signature maintains a list of known propositions and their hash root. Inclusion of such a proposition in the signature may require checking that the term address corresponding to h is owned as a proposition in the ledger. The only way this could have happened is if the term is the axiom of the current theory or was previously proven.) There are three kinds of application and three kinds of abstractions. At the proof level there are applications $(\mathcal{D}\mathcal{E})$ and abstractions $(\lambda_s \mathcal{D})$. These correspond to the elimination and introduction rules for implication. At the term level there are applications $(\mathcal{D}t)$ and abstractions $(\lambda_\alpha \mathcal{D})$. These correspond to the elimination and introduction rules for universal quantification. Finally at the type level there are applications $(\mathcal{D}\alpha)$ and abstractions $(\Lambda \mathcal{D})$. Type level application is the way polymorphic known propositions are applied at specific types. Type level abstraction is the way polymorphic propositions are proven.

As with terms, we omit parentheses assuming application associates to the left and assuming abstraction (binders) have as large a scope as possible.

The corresponding type in the OCaml code is pf. We describe each constructor:

The functions seo_pf and sei_pf serialize and deserialize proof terms.

Again, there are two functions taking a proof term and returning a hash value: hashpf and pf_hashroot. The function hashpf takes a term and returns a hash value obtained by serializing the term to a string, hashing the string, and then hashing the result tagged with 67. This implies hashpf returns an effectively unique hash value for each proof term. The function pf_hashroot

computes a *hash root* similar to the way hash roots for terms are computed. In this case, the hash root for a proof term abbreviation $\hat{\sharp}_h$ is h.

9.4 Publications

There are three kinds of publications: theories, signatures and documents. A theory declares the types of some primitives c_n and gives some axioms. A signature is to be interpreted within a given theory and is intended to make some terms and propositions accessible for use within another publication (a document or another signature). A signature declares some parameters (opaque terms of the form $\sharp_h$) giving the hash root and the simple type, declares some definitions and declares some propositions to be known (either axioms of the theory or previously proven propositions). A document is similar to a signature except proofs of theorems are also allowed. In addition, a document may declare a proposition to be a conjecture.

For all three kinds of publications there is a representation as a list of "items." This list is perhaps best thought of as being in reverse order. The idea is that after one has processed the "rest" of the list, then one has sufficient information to process the "head" of the list.

In practice there is a distinction between the specification of a theory and the theory itself. The same is true of signatures. In essence a theory specification or signature specification corresponds to a list of declarations, where a theory or signature itself is a "compiled" format which other publications may used. This "compiled" format must be stored by every node in order to check later publications. For this reasons, Qeditas currency units must be burned in order to publish a theory or publication. In particular, 21 zerms must be burned for each byte in the serialized representation of the theory or signature. The idea behind a fee of 21 zerms is that since there is an upper bound of 21 million fraenks (21 billion zerms) we can be sure that no more than 1 GB worth of theories and signatures will ever be published.

9.4.1 Theories

A *theory item* is one of the following:

- a declaration of a primitive to have type α,
- a declaration of a definition of type α defined by a term s, or
- a declaration of proposition s as an axiom.

A *theory specification* is a list of theory items.

A *theory* $\mathcal{T}$ is a pair $(\mathcal{P}, \mathcal{A})$ of a list $\mathcal{P}$ of simple types $\alpha_0, \ldots, \alpha_{n-1}$ and a list $\mathcal{A}$ of hash values $\overline{h}$. The idea is that the primitive c_i has the type α_i for $i < n$ and that s is an axiom of the theory if $\mathbf{TR}_\sharp(s)$ is in the list $\overline{h}$.

In the OCaml code the corresponding types are theoryitem, theoryspec and theory. The type theoryitem has three constructors corresponding to the three cases above.

- ThyPrim(α) declares that the primitive c_n has type α. In practice n is the number of ThyPrim theory items in the rest of the theory specification.
- ThyDef(α, s) declares that s has type α and so $\sharp_{\mathbf{TR}_\sharp(s)}$ can be used as an abbreviation.
- ThyAxiom(t) declares that t is a proposition and an axiom of the theory.

As usual, seo_theoryspec and sei_theoryspec serialize and deserialize theory specifications while seo_theory and sei_theory serialize and deserialize theories.

The function theoryspec_theory converts a theory specification to a theory. This is done by extracting the declared types of the first n primitives (see theoryspec_primtps) and by extracting the hash roots of declared axioms (see theoryspec_hashedaxioms). The pair is the intended theory. Note that declared definitions are not used to construct the theory. Definitions in a theory may be required to check the theory specification is valid (e.g., to check that a term for an axiom is, in fact, a proposition).

The function hashtheory computes an optional hash value corresponding to a theory. This hash value will be the identifier for the theory. For the empty theory (no typed primitives and no axioms) the optional hash value will be None. The empty theory is not explicitly stored and cannot be explicitly published. For nonempty theories, the hash value is used to determine the 160-bit location where the theory is stored in the ttree and the publication address where the theory specification is stored in the ledger tree.[2]

The function theoryspec_burncost computes the number of cants that must be burned to publish the theory specification. It does this by computing the underlying theory with theoryspec_theory, serializing the theory, taking the number of bytes of the serialization and multiplying this by 2.1 billion).

9.4.2 Signatures

A *signature item* is one of the following:

- a reference to another signature to import,
- a declaration of a parameter with a given term hash root and given type,
- a declaration of a definition with a certain type and a term which should have this type, or

[2]There is actually no need to store theory publications in the ledger tree since the useful information will be in the ttree. However, it seems simplest to publish theory specifications as a special kind of asset created by a transaction. Such assets are stored in the ledger tree at the given address. Making an exception for theories (and signatures) would be needlessly awkward.

- a declaration of a proposition as known.

A *signature specification* is a list of signature items.

A *global signature* Σ is a pair $(\mathcal{O}, \mathcal{K})$ of lists:

- The first list $\mathcal{O}$ is a list of hash roots, types and optionally terms. That is, each element is $(h, \alpha, \mathsf{None})$ or (h, α, s) where h is the term hash root of a term of type α (the term s if it is given). The intention is that we know the term $\sharp_h$ abbreviates a term of type α. If s is given, we also know $\sharp_h$ can be expanded to be s.
- The second list $\mathcal{K}$ is a list of hash roots and terms. That is, each element is (h, s) where s has hash root h and s is either an axiom of the theory or a previously proven theorem. (Technically, it is h that is the hash root of an axiom or previously proven theorem. Hence we know s is equal to an axiom or previously proven theorem if all hash value abbreviations are expanded.)

A *signature* is a pair of a list of hash values and a global signature. The list of hash values is a list of references to other signatures to import. A signature here is a certain kind of publication which allows easy importation of previous results and should not be confused with cryptographic signatures as discussed in Chapter 6. In the OCaml code we typically write `signat` when referring to cryptographic signatures and `signa` when referring to the kinds of signatures as publications under consideration here.

The corresponding types in OCaml are signaitem (for *signature item*), signaspec (for *signature specification*), gsigna (for *global signature*) and signa (for *signature*). The type sigitem has four constructors corresponding to the four cases above.

- SignaSigna(h) declares the importation of the signature with hash value identifier h. The signature must have been previously published.
- SignaParam(h, α) declares that $\sharp_h$ has type α and can be used as an opaque abbreviation (a "parameter").
- SignaDef(α, s) declares that s has type α and so $\sharp_{\mathbf{TR}_\sharp(s)}$ can be used as an abbreviation.
- SignaKnown(t) declares that t is a proposition which is already known.

As usual, seo_signaspec and sei_signaspec serialize and deserialize signature specifications. seo_signa and sei_signa serialize and deserialize signatures.

The function signaspec_signa compiles a signature specification into a signature. The function signaspec_signas computes the signatures which should be imported by filtering out the declared signatures to import. The function signaspec_trms computes the term hash roots along with their types and optional term definitions by filtering out the parameter and definition declarations. Finally, signaspec_knowns computes the term hash roots and corresponding known propositions by filtering the declared knowns and computing their hash roots.

The function hashsigna computes a hash value to identify the signature. This hash value, combined with the hash value identifier of the intended theory, is used to create both the 160-bit location where the signature is stored in the stree and the publication address where the signature specification is stored in the ledger tree.[3]

The function signaspec_burncost computes the number of cants which must be burned to publish the signature specification. It does this by computing the underlying signature with signaspec_signa, serializing the signature, taking the number of bytes of the serialization and multiplying this by 2.1 billion).

9.4.3 Documents

A *document item* is one of the following:

- a declaration of a signature to import,
- a declaration of a parameter with a term hash root and type,
- a declaration of a definition with a type and a term of this type,
- a declaration of a proposition as known,
- a declaration of a proposition as a conjecture, or
- a declaration of a proposition as a theorem with a proof.

A *document* is a list of document items.

The corresponding OCaml types are docitem and doc. The type docitem has six constructors corresponding to the six cases above.

- DocSigna(h) declares the importation of the signature with hash value identifier h. The signature must have been previously published.
- DocParam(h, α) declares that $\sharp_h$ has type α and can be used as an opaque abbreviation (a "parameter").
- DocDef(α, s) declares that s has type α and so $\sharp_{\mathbf{TR}_\sharp(s)}$ can be used as an abbreviation.
- DocKnown(t) declares that t is a proposition which is already known.
- DocConj(t) declares that t is a proposition to be treated as a conjecture.
- DocPfOf($t, \mathcal{D}$) declares that t is a proposition proven by the proof term $\mathcal{D}$.

[3] As with theories, there is actually no need to store signature publications in the ledger tree since the useful information will be in the stree. Signature specifications are stored in the ledger tree simply to avoid having exceptional cases.

As always, seo_doc and sei_doc serialize and deserialize documents.

There are two functions computing hash values for documents. The function hashdoc computes a unique hash value for each distinct document. This hash value is used to calculate the publication address where the document will be stored in the ledger tree. The function doc_hashroot computes a hash root for a document. This is done by combining hash roots for document items computed by docitem_hashroot.

The purpose of hash roots for documents is to allow for the representation of parts of the document in a way that still allow the hash root to be computed (as with any Merkle tree [19]). The type pdoc is the type of *partial documents*. Partial documents are an approximation of a document nodes can use for proof of storage in a block header.

The functions seo_pdoc and sei_pdoc serialize and deserialize partial documents.

There are again two functions computing hash values. The function hashpdoc computes a unique hash value for each partial document. This is used to compute the hash of the block header in case a partial document is used for proof of storage. The function pdoc_hashroot computes the hash root of the partial document. If a partial document approximates a document, then both will have the same hash root.

9.5 Dependency Checking

There are a number of functions for computing the dependencies of signatures and documents on objects and propositions. In some cases the functions are used to check if the publisher has the right to use the object or proposition. In other cases the functions are used to justify the publisher claiming ownership of an object or proposition. Ownership of an object or proposition allows the control of the right to use the object or proposition later. Finally, some of the functions compute the hash values used to check the ledger tree to see if a term with a given hash root has a given type in a given theory or if a proposition with a given hash root is known in a given theory.

- signaspec_uses_objs collects the parameters (SignaParam) imported into a signature as pairs (h, k) where h is the hash root of the (omitted) term defining the parameter and k is the hash of the type (given by hashtp). It is defined via a tail recursive function signaspec_uses_objs_aux which also ensures the list is duplicate free. This information is needed to check that there is in fact a previously published term with hash root h with the type with hash k in the appropriate theory. It is also needed to check that the publisher has the right to make use of the parameter in this way. The function is used in output_signaspec_uses_objs in the assets module where (h, k) is converted into a pair (h, k') where k' depends on h, k and the (optional) hash value identifier of the theory. The idea is that h is the identifier of the "pure" object (across all theories) while k' is the identifier

of the object in the specific theory. Both h and k' can be owned as objects. In order to use an object as a parameter in a signature, both h and k' must be free to use as objects (without requiring rights). After an object has been published in a signature, every signature and document can freely use everything in the signature simply by importing it.

- signaspec_uses_props collects the hash roots of the known propositions (SignaKnown) imported into a signature. It is defined via a tail recursive function signaspec_uses_props_aux which also ensures the list is duplicate free. This information is needed to check that there is in fact a proposition with hash root h was previously published with a proof in the appropriate theory (or that there is a corresponding axiom of the theory). It is also needed to check that the publisher has the right to make use of the known in this way. The function is used in output_signaspec_uses_props in the assets module where h is converted into a pair (h, k) where k depends on h and the (optional) hash value identifier of the theory. The idea is that h is the identifier of the "pure" proposition (across all theories) while k is the identifier of the proposition in the specific theory. Both h and k can be owned as propositions. In order to use a proposition as a known in a signature, both h and k must be free to use as propositions (without requiring rights). After a proposition has been published in a signature, every signature and document can freely use everything in the signature simply by importing it.

- doc_uses_objs is similar to signa_uses_objs except it collects parameter declarations in documents (DocParam). It is used in output_doc_uses_objs in the assets module to collect (h, k) where h is the identifier for the "pure" object and k is the identifier for the object in the relevant theory. In order to use the object in the document as a parameter, the transaction publishing the document is required to consume a sufficient number of object rights. The cost of obtaining rights (which may range from "free" to "impossible") is determined by the owners of h and k as objects.

- doc_uses_props is similar to signa_uses_objs except it collects parameter declarations in documents (DocKnown). It is used in output_doc_uses_props in the assets module to collect (h, k) where h is the identifier for the "pure" proposition and k is the identifier for the proposition in the relevant theory. In order to use the proposition in the document as a known, the transaction publishing the document is required to consume a sufficient number of proposition rights. The cost of obtaining rights (which may range from "free" to "impossible") is determined by the owners of h and k as propositions.

- doc_creates_objs collects definitions (DocDef) as pairs (h, k) where h is the hash root of the term and k is the hash of the type. This is modified in output_creates_objs in the module assets to be (h, k') where k' depends on h, k and the (optional) hash value identifier of the theory. Again, h is the

identifier for the "pure" object (across all theories) and k' is the identifier for the object in the specific theory. If a document creates objects which are not yet owned as objects, then an owner must be declared with the transaction publishing the document. (The fact that an identifier for an object in a specific theory has an owner as an object implies that the term has the type in the theory. Hence it can later be safely imported as a parameter of the appropriate type in the theory.)

- doc_creates_props collects the hash roots of proven propositions (DocPfOf). Each hash root h is modified in output_creates_props in the module assets to be a pair (h, k) where k depends on the theory. Again, h is the identifier for the "pure" proposition (across all theories) and k is the identifier for the proposition in the theory. If a document creates propositions which are not yet owned as propositions, then an owner must be declared with the transaction publishing the document. (The fact that an identifier for a proposition in a specific theory has an owner as a proposition implies that the proposition has been proven in the theory. Hence the proposition can safely be imported as something known in the theory.)

- doc_creates_neg_props collects the hash roots of propositions p where the negation of p is proven in the document. (Here the negation of p is either $p \to \forall_o x_0$, i.e., p implies false, or $\neg p$ where $\neg$ is $\lambda_o x_0 \to \forall_o x_0$.) In output_creates_neg_props in the module assets combines the hash root with the theory to obtain an identifier for the proposition in the theory. If a document proves the negation of a proposition, then the OwnsNegProp preasset must be published into the corresponding identifier. The purpose of OwnsNegProp is to allow the collection of bounties by proving the negation of a proposition instead of the original proposition. This makes sense as in a consistent theory proving either a proposition or its negation resolves the conjecture.

9.6 Trees of Theories and Signatures

In order to check the correctness of a signature specification or document, the intended theory is required. Likewise, correctness of the new signature specification or document depends on all signatures imported. For this reason, each Qeditas node needs to store every theory and every signature published so far in order to verify the correctness of new signature specifications and documents. Theories and signatures are stored in trees indexed by their associated hash values. The polymorphic type htree in the module htree provides the general infrastructure. The two special cases are ttree and stree. A ttree is simply an htree for storing theories (elements the type theory). An stree is an htree for storing signatures (elements the type signa). Each signature is associated with a specific theory (possibly the empty theory) and can only be imported into signature specification and documents within the same theory. This is enforced

by storing the signature by an index that depends on the hash value identifier of the theory.

- ottree_insert takes an optional ttree, a bit sequence and a theory and returns the ttree resulting from inserting the theory at the location of the bit sequence. Giving None instead of a ttree corresponds to starting with the empty ttree, and so the returned ttree will have exactly one entry. The bit sequence should be the 160-bit hash returned by hashtheory when called on the given theory.

- ostree_insert takes an optional stree, a bit sequence and a signature and returns the stree resulting from inserting the signature at the location of the bit sequence. Giving None instead of a stree corresponds to starting with the empty stree, and so the returned stree will have exactly one entry. The bit sequence should be the 160-bit hash returned by hashopair2 on the optional hash value identifying the intended theory and the hash value returned by hashsigna when called on the given signature.

- ottree_hashroot returns an optional hash value representing the Merkle root of an optional ttree. This value is sometimes called the *theory root* and is in block headers as newtheoryroot. Before the first theory is published, the optional ttree is None and the corresponding hash root is also None.

- ostree_hashroot returns an optional hash value representing the Merkle root of an optional stree. This value is sometimes called the *signature root* and is in block headers as newsignaroot. Before the first signature is published, the optional ttree is None and the corresponding hash root is also None.

- ottree_lookup lookups a theory in an optional ttree given its optional hash value identifier. If the optional hash value identifier is None, no lookup is performed: the empty theory is returned. Otherwise, the hash value is converted to a bit sequence (list of booleans) and htree_lookup attempts to find the theory in the tree. If no entry is found, the exception Not_found is raised.

Note that there is no ostree_lookup. The internal function import_signatures looks up signatures as they are required.

9.7 Substitution and Normalization

Type checking and proof checking depend on substitution and $\beta\eta$-normalization. The functions implementing substitution and $\beta\eta$-normalization are not exposed in the interface, but it is of fundamental importance that they are bug-free. Substitutions and normalization can also be defined for proof terms, but this is not necessary for Qeditas.

Defining substitution requires auxiliary functions to shift de Bruijn indices. For example, suppose we wish to substitute a term $x_0 \to x_1$ for the de Bruijn index x_0 in the term

$$x_0 \to \forall_o x_1 \to x_0$$

– i.e., $x_0 \to (\forall_o(x_1 \to x_0))$. Naively one might expect the result to be

$$(x_0 \to x_1) \to \forall_o x_1 \to (x_0 \to x_1)$$

but this is incorrect. The first occurrence of x_0 is "free" and should be replaced by $x_0 \to x_1$. However, the second occurrence of x_0 corresponds to the $\forall_o$ binder and so is not "free." On the other hand, x_1 inside the scope of the $\forall_o$ binder corresponds to x_0 outside the scope of the $\forall_o$ binder. This might lead one to believe the resulting term should be

$$(x_0 \to x_1) \to \forall_o(x_0 \to x_1) \to x_0.$$

This is, however, also incorrect as the new x_0 and x_1 instead the scope of the $\forall_o$ binder no longer correspond to x_0 and x_1 outside the scope of the $\forall_o$ binder. The correct result would be

$$(x_0 \to x_1) \to \forall_o(x_1 \to x_2) \to x_0.$$

That is, shifting of de Bruijn indices is required when the substitution procedure passes through a binder.

We say an occurrence δ_n of a type variable (in a term or type) is *locally bound for* i if it is beneath j type level binders (Π, Λ or $\hat{\forall}$) where $n < i + j$. Likewise we say an occurrence of x_n of a term variable (in a term) is *locally bound for* i if it is beneath j term level binders (λ_α or $\forall_\alpha$) where $n < i + j$. We will simply say "locally bound" when the intended i is clear.

We write $T \Uparrow_i^j$ as notation for the term or type where each δ_n is shifted to δ_{n+j} unless it is locally bound for i. We write $t \uparrow_i^j$ as notation for the term t where each x_n are shifted to x_{n+j} unless it is locally bound for i. We write $T[i := S]$ as notation for the result of substituting a type or term S for the variable δ_i or x_i in the type or term T.

Returning to the example above, we can outline the necessary calculations in order to compute

$$(x_0 \to \forall_o x_1 \to x_0)[0 := x_0 \to x_1]$$

as follows:

$$\begin{aligned}
&(x_0 \to \forall_o x_1 \to x_0)[0 := x_0 \to x_1] \\
&= (x_0[0 := x_0 \to x_1]) \to ((\forall_o x_1 \to x_0)[0 := x_0 \to x_1]) \\
&= (x_0 \to x_1) \to \forall_o((x_1 \to x_0)[1 := x_0 \to x_1]) \\
&= (x_0 \to x_1) \to \forall_o((x_1[1 := x_0 \to x_1]) \to (x_0[1 := x_0 \to x_1])) \\
&= (x_0 \to x_1) \to \forall_o((x_0 \to x_1) \uparrow_0^1) \to x_0 \\
&= (x_0 \to x_1) \to \forall_o(x_1 \to x_2) \to x_0
\end{aligned}$$

We briefly describe the internal functions for shifting and performing substitutions at the different levels. For these functions to behave as mathematically intended, we need to assume that for each occurrence of $\sharp_h$ h is the hash root of a term where every occurrence of a variable is locally bound (also called a *closed* term).

- tpshift defines $\alpha \Uparrow_i^j$ for shifting type variables in a type α. The defining cases are:
  - $\delta_k \Uparrow_i^j = \delta_k$ if $k < i$. In particular, locally bound type variables are not shifted.
  - $\delta_k \Uparrow_i^j = \delta_{k+j}$ if $k \geq i$.
  - $(\alpha \to \beta) \Uparrow_i^j = (\alpha \Uparrow_i^j \to \beta \Uparrow_i^j)$.
  - $(\Pi\alpha) \Uparrow_i^j = \Pi(\alpha \Uparrow_{i+1}^j)$. The idea in this case is that passing through the Π binder means one more type variable is considered locally bound.
  - $\alpha \Uparrow_i^j = \alpha$ otherwise.
- tmtpshift defines $t \Uparrow_i^j$ for shifting type variables in a term t. The defining cases are:
  - $(st) \Uparrow_i^j = (s \Uparrow_i^j)(t \Uparrow_i^j)$.
  - $(\lambda_\alpha t) \Uparrow_i^j = \lambda_{\alpha \Uparrow_i^j}(t \Uparrow_i^j)$. The term level λ_α binder does not affect which type variables are locally bound.
  - $(s \to t) \Uparrow_i^j = (s \Uparrow_i^j) \to (t \Uparrow_i^j)$.
  - $(\forall_\alpha t) \Uparrow_i^j = \forall_{\alpha \Uparrow_i^j}(t \Uparrow_i^j)$. The term level $\forall_\alpha$ binder does not affect which type variables are locally bound.
  - $(s\alpha) \Uparrow_i^j = (s \Uparrow_i^j)(\alpha \Uparrow_i^j)$.
  - $(\Lambda t) \Uparrow_i^j = \Lambda(t \Uparrow_{i+1}^j)$. The type level Λ binder means one more type variable is locally bound.
  - $(\hat{\forall} t) \Uparrow_i^j = \hat{\forall}(t \Uparrow_{i+1}^j)$. The type level $\hat{\forall}$ binder means one more type variable is locally bound.
  - $t \Uparrow_i^j = t$ otherwise. In particular, $\sharp_h \Uparrow_i^j = \sharp_h$. Assuming h is the hash root of a term s where every type variable is locally bound, then $s \Uparrow_i^j = s$ and h is still the term root of $s \Uparrow_i^j$.
- tmshift defines $t \uparrow_i^j$ for shifting term variables in a term t. The defining cases are:
  - $x_k \uparrow_i^j = x_k$ if $k < i$.
  - $x_k \uparrow_i^j = x_{k+j}$ if $k \geq i$.
  - $(st) \uparrow_i^j = (s \uparrow_i^j)(t \uparrow_i^j)$.

  - $(\lambda_\alpha t) \uparrow_i^j = \lambda_\alpha(t \uparrow_{i+1}^j)$. The term level λ_α binder makes one more term variable locally bound.
  - $(s \to t) \uparrow_i^j = (s \uparrow_i^j) \to (t \uparrow_i^j)$.
  - $(\forall_\alpha t) \uparrow_i^j = \forall_\alpha(t \uparrow_{i+1}^j)$. The term level $\forall_\alpha$ binder makes one more term variable locally bound.
  - $(s\alpha) \uparrow_i^j = (s \uparrow_i^j)\alpha$.
  - $(\Lambda t) \uparrow_i^j = \Lambda(t \uparrow_i^j)$. The type level Λ binder does not change which term level variables are locally bound.
  - $(\hat{\forall} t) \uparrow_i^j = \hat{\forall}(t \uparrow_i^j)$. The type level $\hat{\forall}$ binder does not change which term level variables are locally bound.
  - $t \uparrow_i^j = t$ otherwise. In particular, $\sharp_h \uparrow_i^j = \sharp_h$. Assuming h is the hash root of a term s where every term variable is locally bound, then $s \uparrow_i^j = s$ and h is still the term root of $s \uparrow_i^j$.

- tpsubst defines $\alpha[j := \beta]$ for types α and β. The defining cases are:
  - $\delta_j[j := \beta] = \beta \Uparrow_0^j$. In the special case where $j = 0$ we know $\beta \Uparrow_0^j = \beta$ and so we can simply take $\delta_0[0 := \beta] = \beta$.
  - $\delta_i[j := \beta] = \delta_{i-1}$ if $i > j$. This corresponds to the "removal" of the variable δ_j during the substitution.
  - $(\alpha_1 \to \alpha_2)[j := \beta] = (\alpha_1[j := \beta]) \to (\alpha_2[j := \beta])$
  - $(\Pi\alpha)[j := \beta] = (\Pi(\alpha[j + 1 := \beta]))$
  - $\alpha[j := \beta] = \alpha$ otherwise.

- tmtpsubst defines $s[j := \beta]$ for terms s and types β. The defining cases are:
  - $(st)[j := \beta] = (s[j := \beta])(t[j := \beta])$.
  - $(\lambda_\alpha t)[j := \beta] = \lambda_{\alpha[j:=\beta]}(t[j := \beta])$. The term level λ_α binder does not affect which type variables are locally bound.
  - $(s \to t)[j := \beta] = (s[j := \beta]) \to (t[j := \beta])$.
  - $(\forall_\alpha t)[j := \beta] = \forall_{\alpha[j:=\beta]}(t[j := \beta])$. The term level $\forall_\alpha$ binder does not affect which type variables are locally bound.
  - $(s\alpha)[j := \beta] = (s[j := \beta])\alpha$.
  - $(\Lambda t)[j := \beta] = \Lambda(t[j + 1 := \beta])$. The type level Λ binder means one more type variable is locally bound.
  - $(\hat{\forall} t)[j := \beta] = \hat{\forall}(t[j + 1 := \beta])$. The type level $\hat{\forall}$ binder means one more type variable is locally bound.
  - $t[j := \beta] = t$ otherwise. In particular, $\sharp_h[j := \beta] = \sharp_h$. Assuming h is the hash root of a term s where every type variable is locally bound, then $s[j := \beta] = s$ and h is still the term root of $s[j := \beta]$.

- tmsubst defines $s[j := u]$ for terms s and u. The defining cases are:
  - $x_j[j := u] = u \uparrow_0^j$. In the special case where $j = 0$ we know $u \uparrow_0^j = u$ and so we can simply take $x_0[0 := u] = u$.
  - $x_i[j := u] = x_{i-1}$ if $i > j$. This corresponds to the "removal" of the variable x_j during the substitution.
  - $(st)[j := u] = (s[j := u])(t[j := u])$.
  - $(\lambda_\alpha t)[j := u] = \lambda_\alpha(t[j + 1 := u])$. The term level λ_α binder makes one more term variable locally bound.
  - $(s \to t)[j := u] = (s[j := u]) \to (t[j := u])$.
  - $(\forall_\alpha t)[j := u] = \forall_\alpha(t[j + 1 := u])$. The term level $\forall_\alpha$ binder makes one more term variable locally bound.
  - $(s\alpha)[j := u] = (s[j := u])\alpha$.
  - $(\Lambda t)[j := u] = \Lambda(t[j := u])$. The type level Λ binder does not change which term level variables are locally bound.
  - $(\hat{\forall} t)[j := u] = \hat{\forall}(t[j := u])$. The type level $\hat{\forall}$ binder does not change which term level variables are locally bound.
  - $t[j := u] = t$ otherwise. In particular, $\sharp_h[j := u] = \sharp_h$. Assuming h is the hash root of a term s where every term variable is locally bound, then $s[j := u] = s$ and h is still the term root of $s[j := u]$.

Next we need to say what it means for a type variable δ_j to be *free* in a type or term, and what it means for a term variable x_j to be *free* in a term. There are three relevant definitions:

- The function free_tpvar_in_tp_p determines if a type variable δ_j is *free* in a type α. The definition is by recursion on α and the j must be increased by 1 in the Π binder case to account for the new locally bound variable.
- The function free_tpvar_in_tm_p determines if a type variable δ_j is *free* in a term t. The definition is by recursion on t and the j must be increased by 1 in the Λ and $\hat{\forall}$ binder cases.
- The function free_in_tm_p determines if a term variable x_j is *free* in a term t. The definition is by recursion on t and the j must be increased by 1 in the λ_α and $\forall_\alpha$ binder cases.

We can now turn to $\beta\eta$-normalization. We begin by considering four kinds of *redexes* and their corresponding *reducts*. Normalization is performed by reducing redexes to their reducts until no more redexes remain. A theorem of various type theories is that normalization terminates in a unique normal form for well-typed terms, and that is true for the type theory under consideration here.

- A term of the form $(\lambda_\alpha s)t$ is a *term level β-redex* with reduct $s[0 := t]$.

- A term of the form $(\Lambda s)\alpha$ is a *type level β-redex* with reduct $s[0 := \alpha]$.

- A term of the form $\lambda_\alpha(sx_0)$ where x_0 is not free in s is a *term level η-redex* with reduct $s \uparrow_0^{-1}$. (The shift of term variables by -1 is required since s was in the scope of one term level binder λ_α which is removed from the reduct.)

- A term of the form $\Lambda(s\delta_0)$ where δ_0 is not free in s is a *type level η-redex* with reduct $s \Uparrow_0^{-1}$. (The shift of type variables by -1 is required since s was in the scope of one type level binder Λ which is removed from the reduct.)

A term is *normal* if it has no redexes. The function tm_norm_p checks if a term is normal. In theory specifications, signature specifications and documents all definitions, knowns, conjectures and theorems are required to be normal. The exception NonNormalTerm is raised if this requirement is violated.

The normalization procedure is tm_beta_eta_norm. It proceeds by repeatedly calling tm_beta_eta_norm_1. In simple terms, the function tm_beta_eta_norm_1 recursively traverses a term reducing each redex it finds and returning the reduced term along with a boolean indicating if at least one reduction was performed. If no reductions were performed, then the term is normal and the procedure ends.

In certain examples, β-normalization (and hence $\beta\eta$-normalization) leads to large terms and can require an unrealistic number of β-reductions. This problem is dealt with by having resource bounds represented internally by beta_count and term_count. Before a signature specification or document is checked, these resource bounds are reset (by reset_resource_limits). (Checking a theory specification requires no β-reductions.) There are $200,000$ beta reductions and 10 million term traversal steps allowed per signature specification or document. Each β-reduction step decrements beta_count by one. If beta_count reaches 0, then the exception BetaLimit is raised. For each recursive call of a shift or substitution function decrements term_count. If term_count reaches 0, then the exception TermLimit is raised. If either of these exceptions are thrown, it essentially means that checking the signature specification or document is too resource intensive and it cannot be published in its current form. If this occurs, a possible solution is to factor the publication into multiple publications.

It is worth noting that some well-known ill-typed terms do not have a normal form. For example, $(\lambda_o x_0 x_0)(\lambda_o x_0 x_0)$ is a term level β-redex with itself as a reduct. Without resource bounds, calling tm_beta_eta_norm with this term would result in an infinite loop. With the resource bound, BetaLimit would be raised. In practice, tm_beta_eta_norm should never be called with such a term since during the checking of publications tm_beta_eta_norm is only called with terms which are already known to be well-typed.

We write $[s]^\downarrow$ for the $\beta\eta$-normal form of s, assuming it exists.

$$\frac{i < v}{v \vdash \delta_i \text{ stp}} \qquad \frac{v \vdash \alpha \text{ stp} \qquad v \vdash \beta \text{ stp}}{v \vdash \alpha \to \beta \text{ stp}} \qquad \frac{v \vdash \alpha \text{ stp}}{v \vdash \alpha \text{ ptp}} \qquad \frac{v+1 \vdash \alpha \text{ ptp}}{v \vdash \Pi\alpha \text{ ptp}}$$

Figure 9.1: Rules for validity of types

9.8 Type Checking and Proof Checking

We now turn to the most important functions: those which check that a type is valid, check that a term has a type, check that a term is a proposition, and check that a proof term is a proof of a proposition.

Checking attempts typically either succeed (possibly returning some information) or raise an exception. The exception CheckingFailure is raised if checking fails. One of the exceptions BetaLimit or TermLimit is raised if one of the corresponding resource bounds is reached.

All the properties defined will be relative to a *type context*. Since type variables are represented as de Bruijn indices, the *type context* can be taken to simply be a non-negative integer v.

We say a type α is *valid as a simple type* in type context v (and write $v \vdash \alpha$ stp) if it contains no occurrence of Λ and every type variable δ_i satisfies $i < v$. The function check_tp performs this check. A type α is *valid as a polymorphic type* in type context v (and write $v \vdash \alpha$ ptp) if if it is of the form

$$\underbrace{\Lambda \cdots \Lambda}_{m} \beta$$

where β is valid as a simple type in type context $v+m$. The function check_ptp performs this check.

Properties such as these are often defined using rules. One can often understand the functions checking the properties better by comparing them to such rules. Rules defining $v \vdash \alpha$ stp and $v \vdash \alpha$ ptp are given in Figure 9.1.

The notion of types being valid is independent of the theory or a signature. It would make sense to restrict types to only mention base types mentioned in the current theory, but there is no strong reason to do so. For simplicity, we ignore the theory when asking if a type is valid.

The property of when a term has a type depends on both a type context and a *term context*. A *term context* Γ is a list of types $\alpha_0, \ldots, \alpha_{m-1}$ giving the types of the term level de Bruijn indices in the current context. We write $\Gamma \Uparrow_i^j$ for the term context $\alpha_0 \Uparrow_i^j, \ldots, \alpha_{m-1} \Uparrow_i^j$ when Γ is $\alpha_0, \ldots, \alpha_{m-1}$.

We now define $\Sigma; v; \Gamma \vdash t : \alpha$ – i.e., when a term t has type α in a signature Σ, type context v and term context Γ. The definition also depends on the current theory, but we will leave this implicit in the notation. Suppose the current theory assings the types $\gamma_0, \ldots, \gamma_{n-1}$ to the first n primitives. This information is needed to know $c_i : \gamma_i$ for $i < n$. (The corresponding function is tp_of_prim.) Rules defining $\Sigma; v; \Gamma \vdash t : \alpha$ are given in Figure 9.2.

$$\frac{\Gamma = \alpha_0, \ldots, \alpha_{m-1} \qquad i < m}{\Sigma; v; \Gamma \vdash x_i : \alpha_i} \qquad \frac{i < n}{\Sigma; v; \Gamma \vdash c_i : \gamma_i}$$

$$\frac{\Sigma = (\mathcal{O}, \mathcal{K}) \qquad (h, \alpha, d) \in \mathcal{O}}{\Sigma; v; \Gamma \vdash \sharp_h : \alpha} \qquad \frac{\Sigma; v; \Gamma \vdash s : \alpha \to \beta \qquad \Sigma; v; \Gamma \vdash t : \alpha}{\Sigma; v; \Gamma \vdash st : \beta}$$

$$\frac{\Sigma; v; \alpha, \Gamma \vdash s : \beta}{\Sigma; v; \Gamma \vdash \lambda_\alpha s : \alpha \to \beta} \qquad \frac{\Sigma; v; \Gamma \vdash s : o \qquad \Sigma; v; \Gamma \vdash t : o}{\Sigma; v; \Gamma \vdash s \to t : o}$$

$$\frac{\Sigma; v; \alpha, \Gamma \vdash s : o}{\Sigma; v; \Gamma \vdash \forall_\alpha s : o} \qquad \frac{\Sigma; v; \Gamma \vdash s : \Pi\alpha \qquad v \vdash \beta \text{ stp}}{\Sigma; v; \Gamma \vdash s\beta : \alpha[0 := \beta]} \qquad \frac{\Sigma; v+1; \Gamma \Uparrow_0^1 \vdash s : \alpha}{\Sigma; v; \Gamma \vdash \Lambda s : \Pi\alpha}$$

Figure 9.2: Rules for typing terms

$$\frac{\Sigma; v; \cdot \vdash s : o}{\Sigma; v \vdash s \text{ polyprop}} \qquad \frac{\Sigma; v+1 \vdash s \text{ polyprop}}{\Sigma; v \vdash \hat{\forall} s \text{ polyprop}}$$

Figure 9.3: Rules for when terms are polymorphic propositions

There are two mutually recursive functions extr_tpoftm and check_tpoftm to check $\Sigma; v; \Gamma \vdash t : \alpha$. The function extr_tpoftm is not given the type α and extracts the type α, returning it upon success. The function check_tpoftm is given the α and ensures it matches the extracted type.

In simple type theory a term is often called a "proposition" in a given context when it has type o This will be true here, but we will have additional polymorphic propositions of the form

$$\underbrace{\hat{\forall} \cdots \hat{\forall}}_{m} s$$

where s has type o in the empty context. The restriction considering such propositions in the empty context is to ensure that all $\hat{\forall}$ type binders occur before term level binders. We can define $\Sigma; v \vdash s$ polyprop meaning s is a *polymorphic proposition* (under the signature Σ and type context v) by the rules in Figure 9.3. We then define $\Sigma; v; \Gamma \vdash s$ prop meaning s is a *proposition* (under the signature Σ, type context v and term context Γ) to mean $\Sigma; v \vdash s$ polyprop or $\Sigma; v; \Gamma \vdash s : o$.

The function check_polyprop implements the check for polymorphic propositionhood and check_prop implements the check for propositionhood.

In order to determine if a proof term proves a given proposition, it is necessary to consider the proposition up to $\beta\eta$-reductions and expansion of definitions in Σ. Expansion of definitions is sometimes called δ-reduction. That is,

if $\Sigma = (\mathcal{O}, \mathcal{K})$ is a global signature and $(h, \alpha, s) \in \mathcal{O}$, then $\sharp_h$ is a δ-redex with δ-reduct s. We write $[s]^{\Sigma}$ for the δ-normal form of s, assuming it exists. (If there were cycles in the definitions in Σ, then δ-reduction would not terminate. In practice, signatures will not have such cycles.) Note that δ-reduction may (and often will) introduce new β-redexes. However, $\beta\eta$-reduction never introduces new δ-redexes. Hence we can compute the $\beta\eta\delta$-normal form of s by first computing $[s]^{\Sigma}$ and then computing the $\beta\eta$-normal form. We denote this $\beta\eta\delta$-normal form by $[s]^{\Sigma\downarrow}$. In the special case where we know s is δ-normal already, then $[s]^{\Sigma\downarrow}$ is the same as $[s]^{\downarrow}$ (the $\beta\eta$-normal form of s).

An easy way to consider terms up to $\beta\eta\delta$-reduction is to simply always normalize the terms. This is what the code does when trying to determine if a proof term proves a proposition.

The function tm_delta_norm computes $[s]^{\Sigma}$ and the function tm_beta_eta_delta_norm computes $[s]^{\Sigma\downarrow}$.

The property of when a proof term is a proof of a given proposition depends on a type context, a term context and a *hypothesis context*. A *hypothesis context* Φ is a list $\rho_0, \ldots, \rho_{k-1}$ of terms giving the current assumed hypotheses (proof level de Bruijn indices in the proof term). We will only work with hypothesis contexts where each hypothesis ρ is $\beta\eta\delta$-normal. We will also ensure that the terms are of type o (and not, for example, more general polymorphic propositions).

Let Φ be $\rho_0, \ldots, \rho_{k-1}$. We write $\Phi \uparrow_i^j$ for the hypothesis context $\rho_0 \uparrow_i^j, \ldots, \rho_{k-1} \uparrow_i^j$ when Φ is $\rho_0, \ldots, \rho_{k-1}$.

We now define when a proof term proves a proposition, given a signature and appropriate contexts. We denote this relation by $\Sigma; v; \Gamma; \Phi \vdash \mathcal{D} : t$ and define it by the rules in Figure 9.4. Note that in each rule, we ensure that the proposition t is $\beta\eta\delta$-normal (assuming every proposition in Φ is $\beta\eta\delta$-normal). Also note that proof terms for polymorphic propositions which are not of type o (i.e., have at least one $\hat{\forall}$ binder) must either be known $\Box_h$ or must be proven in the empty term and hypothesis contexts.

Like with type checking, there are two mutually recursive functions extr_propofpf and check_propofpf to check $\Sigma; v; \Gamma; \Phi \vdash \mathcal{D} : t$. In the case of extr_propofpf no proposition t is given. If the given proof term does prove a proposition t, this $\beta\eta\delta$-normal t is returned. Otherwise, the exception CheckingFailure (or one of the resource bound exceptions) is raised. In the case of check_propofpf a $\beta\eta\delta$-normal proposition t is given and is checked to be equal to the result returned by extr_propofpf. If the two are not equal, CheckingFailure is raised.

9.9 Publication Checking

We can now turn to the main functions exported by the mathdata module: the functions for checking theory specifications, signature specifications and documents. In each case, if the publication is checked to be correct, a global signature is returned. If a theory specification is checked to be correct, then a theory is also returned. If a publication is determined not to be correct, an exception is

$$\frac{\Phi = \rho_0, \ldots, \rho_{k-1} \qquad j < k}{\Sigma; v; \Gamma; \Phi \vdash \mathbf{H}_j : \rho_j} \qquad \frac{\Sigma = (\mathcal{O}, \mathcal{K}) \qquad (h, s) \in \mathcal{K}}{\Sigma; v; \Gamma; \Phi \vdash \Box_h : [s]^{\Sigma\downarrow}}$$

$$\frac{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D} : \forall_\alpha s \qquad \Sigma; v; \Gamma; \Phi \vdash t : \alpha}{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D}t : [s[0 := [t]^\Sigma]]^\downarrow}$$

$$\frac{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D} : s \to t \qquad \Sigma; v; \Gamma; \Phi \vdash \mathcal{E} : s}{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D}\mathcal{E} : t}$$

$$\frac{v \vdash s \text{ stp} \qquad \Sigma; v; \alpha, \Gamma; \Phi \uparrow_0^1 \vdash \mathcal{D} : s}{\Sigma; v; \Gamma; \Phi \vdash \lambda_\alpha \mathcal{D} : \forall_\alpha s}$$

$$\frac{\Sigma; v; \Gamma \vdash s : o \qquad \Sigma; v; \Gamma; [s]^{\Sigma\downarrow}, \Phi \vdash \mathcal{D} : t}{\Sigma; v; \Gamma; \Phi \vdash \lambda_s \mathcal{D} : [s]^{\Sigma\downarrow} \to t} \qquad \frac{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D} : \hat{\forall} s \qquad v \vdash \beta \text{ stp}}{\Sigma; v; \Gamma; \Phi \vdash \mathcal{D}\beta : s[0 := \beta]}$$

$$\frac{\Sigma; v + 1; \cdot; \cdot \vdash \mathcal{D} : s}{\Sigma; v; \cdot; \cdot \vdash \Lambda \mathcal{D} : \hat{\forall} s}$$

Figure 9.4: Rules for when a proof proves a proposition

raised. Some of these exceptions have been mentioned above: BetaLimit (too many β-reductions), TermLimit (too many large terms), NonNormalTerm (a term in a publication was not $\beta\eta$-normal) and CheckingFailure (either type checking or proof checking failed). In addition, there are three more exceptions:

- NotKnown is an exception indicating an attempt to import proposition as previously known failed.
- UnknownTerm is an exception indicating an attempt to import an object by the hash root of the term failed.
- UnknownSigna is an exception indicating that an attempt was made to import an unknown signature.

Signature specifications and documents may import previous signatures and this is handled by import_signatures. An extra value imported prevents importing of signatures multiple times.

There are three functions for checking publications:

- check_theoryspec checks a theory specification and (upon success) returns a theory and a global signature. The empty theory specification returns the empty theory and the empty global signature. Suppose the theory specification is nonempty. We recursively call the procedure on the rest of the specification to obtain a theory $\mathcal{T} = (\mathcal{P}, \mathcal{A})$ and global signature $\Sigma = (\mathcal{O}, \mathcal{K})$. The head of the specification is then handled as follows:

  - ThyPrim(α): check that $0 \vdash \alpha$ ptp (α is valid as a polymorphic type in type context 0) and return the theory $(\mathcal{P}', \mathcal{A})$ where $\mathcal{P}'$ is $\mathcal{P}, \alpha$ and global signature Σ. This declares the type of the next primitive c_n.
  - ThyDef(α, s): check that s is $\beta\eta$-normal, $0 \vdash \alpha$ ptp (α is valid as a polymorphic type in type context 0) and that $\Sigma; 0; \cdot \vdash s : \alpha$ (s has type α) and return the theory $\mathcal{T}$ and the global signature $(\mathcal{O}', \mathcal{K})$ where $\mathcal{O}'$ is $\mathcal{O}$ with the additional entry $(\mathbf{TR}_\sharp(s), \alpha, s)$. That is, the global signature has enough information to recover the type of s and the term s from the hash root $\mathbf{TR}_\sharp(s)$.
  - ThyAxiom(t) check that t is a $\beta\eta$-normal proposition (in the type context 0 and empty term context). Let k be the hash root $\mathbf{TR}_\sharp(t)$. Let $\mathcal{A}'$ be $\mathcal{A}$ with the additional entry k (to record the new axiom of the theory). Let $\mathcal{K}'$ be $\mathcal{K}$ with the additional entry (k, t) (to record that $\square_k$ is a proof of t). Return the theory $(\mathcal{P}, \mathcal{A}')$ and global signature $(\mathcal{O}, \mathcal{K}')$.

- check_signaspec checks a signature specification and (upon success) returns a global signature. This is performed within a fixed theory $\mathcal{T} = (\mathcal{P}, \mathcal{A})$. The work is done by an auxiliary function check_signaspec_rec which keeps up with which signatures have been imported. We can define the recursive procedure as follows: The empty signature specification returns the empty global signature. Otherwise, we recursively call the procedure on the rest of the document to obtain a global signature $\Sigma = (\mathcal{O}, \mathcal{K})$. Then we handle each of the signature item cases as follows:

  - SignaSigna(h): import the signature identified by h. If no signature identified by h is found, then raise UnknownSigna. Otherwise, the result is a signature $\Sigma' = (\mathcal{O}', \mathcal{K}')$ where $\mathcal{O} \subseteq \mathcal{O}'$ and $\mathcal{K} \subseteq \mathcal{K}'$. (Here $\subseteq$ means each entry in the first list is an entry in the second.)
  - SignaParam(h, α): check that $0 \vdash \alpha$ ptp and that the term with hash root h is known to have type α. This second check is performed by calling tm_tp_p with a generic test gvtp, the signature, the theory identifier, h and α. The test can succeed by noting h is already declared in the signature to have type α or by verifying gvtp when called on the theory identifier, h and α. Later when check_signaspec is used, gvtp is instantiated by a test to determine if a certain term address (formed from the theory identifier, h and α) is owned as an object. Assuming these checks succeed, return the signature $(\mathcal{O}', \mathcal{K})$ where $\mathcal{O}'$ is $\mathcal{O}$ with the additional entry $(h, \alpha, \mathsf{None})$.
  - SignaDef(α, s): check s is $\beta\eta$-normal, $0 \vdash \alpha$ ptp and $\Sigma; 0; \cdot \vdash s : \alpha$. Let h be the hash root $\mathbf{TR}_\sharp(s)$. If s is $\sharp_h$, then return Σ (since recording h as defined by $\sharp_h$ would be cyclic). If s is not $\sharp_h$, then return $(\mathcal{O}', \mathcal{K})$ where $\mathcal{O}'$ is $\mathcal{O}$ with the additional entry (h, α, s).
  - SignaKnown(t): check t is $\beta\eta$-normal and that $\Sigma; 0; \cdot \vdash t$ prop. Let k be the hash root $\mathbf{TR}_\sharp(t)$. Assuming k is a the hash root of a known

proposition, return $(\mathcal{O}, \mathcal{K}')$ where $\mathcal{K}'$ is $\mathcal{K}$ with the additional entry (k, t). In order for k to be the hash root of a known proposition, we check that either $k \in \mathcal{A}$ (so k is the hash root of an axiom of the theory) or the test known_p passes using a generic test gvkn of the signature, theory identifier and k holds. The test known_p passes if either $(k, s) \in \mathcal{K}$ for some s (so k known in the signature) or gvkn holds for the theory identifier and k. Later when check_signaspec is used, gvkn is instantiated by a test to determine if a certain term address (formed from the theory identifier and k) is owned as a proposition.

- check_doc checks a document and (upon success) returns a global signature. The work is done by an auxiliary function check_doc_rec which keeps up with which signatures have been imported. We can define the recursive procedure as follows: The empty document returns the empty global signature. Otherwise, we recursively call the procedure on the rest of the document to obtain a global signature $\Sigma = (\mathcal{O}, \mathcal{K})$. Then we handle each of the document item cases as follows:
  - DocSigna(h) is handled like SignaSigna in the case of signature specifications.
  - DocParam(h, α) is handled like DocParam in the case of signature specifications.
  - DocDef(α, s) is handled like DocDef in the case of signature specifications.
  - DocKnown(t) is handled like DocKnown in the case of signature specifications.
  - DocConj(t): check that $\Sigma; 0; \cdot \vdash t$ prop and return the same global signature Σ. Conjectures do not change the signature and cannot be used in the rest of the document.
  - DocPfOf$(t, \mathcal{D})$: check that $\Sigma; 0; \cdot \vdash t$ prop and $\Sigma; 0; \cdot \vdash \mathcal{D} : [t]^{\Sigma\downarrow}$. Return $(\mathcal{O}, \mathcal{K}')$ where $\mathcal{K}'$ is $\mathcal{K}$ with $(\mathbf{TR}_\sharp(t), t)$. That is, t (identified by its hash root) is added to the list of known propositions.

Chapter 10

Assets and Transactions

The module assets defines a type asset. It also contains code to support the inputs and outputs of transactions. The module tx defines a type tx of transactions and a type stx of signed transactions, as well as code for checking the validity of transactions and their signatures. "Validity" of a transaction is a weak form of correctness that a transaction must satisfy before asking if it is supported by the current ledger.

10.1 Assets

An asset consists of four pieces of information: a hash value (the *asset id*), a 64-bit integer giving the block in which the asset was published (the *birthday*), an obligation (indicating who controls the asset) and a preasset (determining the kind of asset). In the case of an asset in the initial distribution, the asset id is the 160-bit hash value corresponding to the p2pkh or p2sh address. (Since no p2pkh or p2sh addresses in the snapshot had the same 160-bit address, these asset ids are unique.) In the case of an asset created as the output of a transaction, the asset id is formed from hashing the transaction id paired with the index of the output creating the identifier. Assets in the initial distribution are given birthday 0 and the first block will be considered to have block height 1. Obligations and preassets are described below.

Note: Unit tests for the assets module are in `assetsunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

Note: The Coq module Assets is intended to correspond to assets. There are Coq types preasset, obligation and asset corresponding to the types with the same names defined in OCaml. One difference is that in the OCaml code an obligation also keeps a boolean indicating if the asset is a reward since this was needed to implement forfeiture of rewards in case a staker double signs.

Readers can examine the formal properties proven in Assets to have a better idea of what properties corresponding OCaml functions should satisfy. For more information, see [32], although preassets in the version described there are restricted to currency units.

10.1.1 Obligations

We first consider the type obligation:

```
type obligation = (payaddr * int64 * bool) option
```

Note that an obligation may be empty, which usually means the address that *holds* the asset can spend it (here *holds* refers to the address in the ledger tree where the asset is stored). In case an obligation is not empty, it consists of a triple (α, n, r). Here α is a pay address (a p2pkh address or p2sh address) which must sign in order to spend the asset. (The holder of the asset is the one who can use the asset to stake. Hence obligations can be used to "loan" an asset to a staker without giving the staker the ability to "spend" the asset.) The integer n is the earliest block height at which the asset can be spent. (The intention here is to "lock" an asset for a period of time. Such "locked" assets are given preference when staking.) The boolean r indicates if the asset is a reward for staking a block.[1] Rewards are considered special in the sense that they can be forfeited in the first 6 blocks after the reward is issued, if the issuer provably double signs within the next 6 blocks.

10.1.2 Categories of Preassets and Assets

There are 11 kinds of *assets*, as determined by the corresponding *preasset*: currency units, bounties, object ownership, proposition ownership, negated proposition ownership, object rights, proposition rights, markers, theory publications, signature publications and document publications. The type preasset consists of the following 11 corresponding constructors:

- Currency(n) represents n cants of currency units, where n is a 64-bit integer. A *cant* is the smallest currency unit considered in Qeditas.[2] Currency units can be transfered by fulfilling the appropriate obligation (which usually simply means signing the transaction spending the asset with an appropriate private key).

- Bounty(n) represents n cants as a bounty on a proposition. Bounties are held at term addresses, specifically at the term address of a proposition in a theory. A bounty can be spent (and transformed into currency) by the proposition owner or negated proposition owner. Typically neither the proposition nor its negation have been proven in the theory (and so it has

[1] Philosophically, this should not be part of the "obligation," but the reward indicator was added late and this was a simple way to include it.

[2] The word "cants" is pronounced with a hard c as it is derived from the name Cantor.

neither a proposition owner nor a negated proposition owner) when the bounty is placed.[3] If someone publishes a document in which the proposition is proven, the publisher declares the proposition owner. Likewise, if someone publishes a document in which the negation of the proposition is proven, the publisher declares the negated proposition owner. In either case, the new owner (presumably the publisher) can then collect the bounty.

- OwnsObj(α, p) corresponds to a declaration of object ownership of a term (either a pure term or a term in a theory). The α is a pay address and the p is an optional 64 bit integer. The actual object owner is determined by the obligation of the corresponding asset (and so may or may not be α). The address α is intended as an address others can pay in order to purchase rights to use the object (as an imported parameter) in future documents. The optional value p gives the price (in cants) to purchase one right. If p is 0, then the object can be freely used (without a need to purchase rights). If p is None, then the object cannot be used in this way at all (and rights cannot be purchased). (The object can always be used in a new document by repeating the definition.)

- OwnsProp(α, p) corresponds to a declaration of proposition ownership of a term (either a pure term or a term in a theory). The α is a pay address and the p is an optional 64 bit integer. The actual proposition owner is determined by the obligation of the corresponding asset (and so may or may not be α). The address α is intended as an address others can pay in order to purchase rights to use the proposition (as an imported known) in future documents. The optional value p gives the price (in cants) to purchase one right. If p is 0, then the proposition can be freely used (without a need to purchase rights). If p is None, then the proposition cannot be used at all (and rights cannot be purchased). (The proposition can always be used in a new document by reproving it.)

- OwnsNegProp corresponds to a declaration of a negated proposition ownership of a term in a theory. Again, the "owner" is determined by the corresponding obligation. This kind of asset is only to facilitate the collection of a bounty by disproving a conjecture with a bounty.

- RightsObj(α, n) corresponds to the right to use the object with term address n times. Some or all of these rights will be consumed when publishing a document which imports the object as a parameter (omitting the definition). Note that to use objects within a theory, rights may be required for the pure object (independent of the theory) and for the object within the theory. These are two different term addresses.

[3]The "proposition owner" is determined by the (nonempty) obligation at the proposition ownership asset held at the term address, if there is such an asset. Likewise, the "negated proposition owner" is determined by the obligation at the negated proposition ownership asset held at the term address, if there is such an asset.

- RightsProp(α, n) corresponds to the right to use the proposition with term address α n times. Some or all of these rights will be consumed when publishing a document which imports the proposition as a known (without proof). Note that to use propositions within a theory, rights may be required for the pure proposition (independent of the theory) and for the proposition within the theory. These are two different term addresses.

- Marker is for part of the protocol for publishing a document. A publication address is determined by the (privately known) publication with a (privately known) nonce. A marker must be at the publication address (as an *intention to publish*) for 144 blocks (see intention_minage) before the actual publication can be published. The idea is that the true author of the document publishes the marker roughly a day before revealing the publication itself. The publication is revealed in the transaction publishing it. At that point, a plagiarist could take the publication, compute a new nonce, publish a new marker and then try to publish their copy. However, they would need to wait at least 144 blocks before their copied version could be published. By that time, the original publication should already be published. The order of publication is important since this may determine ownership of newly defined objects and newly proven propositions.

- TheoryPublication(α, ν, τ) is a preasset for publishing a theory specification (theoryspec) τ. The pay address α identifies the author (possibly "publisher" is more accurate) and the corresponding transaction creating such an asset must be signed by α. The hash value ν is a nonce to determine the publication address for the marker which must be published 144 blocks before the publication can be published.

- SignaPublication(α, ν, h, Σ) is a preasset for publishing a signature specification (signaspec) Σ. The pay address α identifies the author and the corresponding transaction creating such an asset must be signed by α. The hash value ν is a nonce to determine the publication address for the marker which must be published 144 blocks before the publication can be published. The optional hash value h identifies the theory in which the signature belongs. An object or proposition can only be included in a signature if no rights are required to use the object or proposition. (The empty theory is identified by giving None for h.)

- DocPublication(α, ν, h, Δ) is a preasset for publishing a document (doc) Δ. The pay address α identifies the author and the corresponding transaction creating such an asset must be signed by α. The hash value ν is a nonce to determine the publication address for the marker which must be published 144 blocks before the publication can be published. The optional hash value h identifies the theory in which the signature belongs. (The empty theory is identified by giving None for h.)

The type asset of assets is now simply defined as a product.

```
type asset = hashval * int64 * obligation * preasset
```

The functions assetid, assetbday, assetobl and assetpre extract the components from the asset.

10.1.3 Types for Transaction Inputs and Outputs

The inputs of transactions will be pairs of addresses and asset identifiers (hash values) of assets held at these addresses. The type addr_assetid plays the role of a transaction input and is defined as follows:

```
type addr_assetid = addr * hashval
```

The outputs of transactions are triples of addresses, obligations and preassets. (The asset identifier is determined by the transaction itself and the birthday is determined by the block height in which the transaction is included.) The type addr_preasset plays the role of a transaction output and is defined as follows:

```
type addr_preasset = addr * (obligation * preasset)
```

The inputs and outputs of a transaction can be elaborated into a pair of an address with an asset in certain situations. While checking a transaction is supported the input assets are looked up from the ledger tree using the asset identifier. A transaction output gives the obligation and preasset. When a transaction is being included in a block at a given height, we know the birthday and can use this (along with the asset identifier which is derived from the transaction) to form the asset. The type addr_asset is included to represent such an elaborated input or output.

```
type addr_asset = addr * asset
```

10.1.4 Functions

The functions hashobligation hashes an obligation (returning None for the None obligation). The functions hashpreasset, hashasset, hash_addr_assetid, hash_addr_preasset and hash_addr_asset hash the corresponding types.

As usual, there are functions for serializing and deserializing elements of these types: seo_obligation, sei_obligation, seo_preasset, sei_preasset, seo_asset, sei_asset, seo_addr_assetid, sei_addr_assetid, seo_addr_preasset, sei_addr_preasset, seo_addr_asset and sei_addr_asset.

The purpose of the remaining functions exported by the assets module are as follows:

- new_assets takes a birthday b, an address α, an addr_preasset list (transaction outputs), a hash value (which should be the hash of the transaction) and an output index (which should be 0 in the initial call) and returns a list of assets which would be put into address α if the transaction is published at block height b. The transaction hash and output index are used to compute the asset ids.

- remove_assets takes an asset list and a list of asset identifiers (the "spent list") and returns the asset list after removing the assets with ids in the spent list.

- get_spent takes an address α and an addr_assetid list (transaction inputs) and returns a list of asset ids being spent from the given address.

- add_vout is similar to new_assets except it is not specific to an address.[4] It takes a birthday b, a hash value (which should be the hash of the transaction), an addr_preasset list (transaction outputs) and an output index (which should be 0 in the initial call), and returns an addr_asset list consisting of the fully elaborated output assets (assuming the transaction is published at block height b).

- preasset_value takes a preasset, a birthday (which should be the birthday of the corresponding asset) and a block height and returns the optional number of cants that a preasset is worth. Only currency units and bounties are worth cants. For other preassets, None is returned. If the preasset is a bounty preasset with v cants, then the value is v cants. If the birthday is not 0 (so the preasset was not part of the initial distribution) and the preasset is a currency preasset with v cants, then the value is v cants. Currency assets from the initial distribution (with birthday 0) are treated in a special manner. In particular, their value will halve along with the block rewards. Suppose the birthday is 0 and the preasset is a currency asset with v cants. Until block height $280,000$, the preasset is worth v cants. Block height $280,000$ is when the second reward halving occurs and should occur roughly 5 years after the network begins running. For the next $210,000$ blocks after block height $280,000$ the value is divided in half (rounding down). The value continues to be divided in half each $210,000$ blocks from that point on until block height $11,410,000$ at which point all currency preassets with birthday 0 have value 0 cants. Block height $11,410,000$ should occur after roughly 200 years.[5]

- asset_value returns the value of the underlying preasset.

- asset_value_sum returns the sum of the value of a list of assets (where None is counted as 0).

- output_signaspec_uses_objs takes an addr_preasset list (transaction outputs) and returns a list of pairs of term addresses. For each object imported as a parameter by a signature specification being published as one of the outputs, (α, β) will be on the output list where α is the term address given by the hash root h[6] of the term which was used to define the object and β is the term address given by hashing h with the type of the object and

[4] It might make sense to delete one of these functions in favor of the other.

[5] The code for halving the value of the unclaimed initial distribution was added by Trent Russell in early 2016, at his suggestion.

[6] Recall that term addresses are actually hash values, so $\alpha = h$.

with the identifier of the current theory (and then tagging this with 32 to avoid accidental collision). If an object is imported by multiple different signatures being published, then the pair will be on the list multiple times. The information is obtained by calling signaspec_uses_objs on appropriate preassets.

- output_signaspec_uses_props takes an addr_preasset list (transaction outputs) and returns a list of pairs of term addresses. For each proposition imported as a known by a signature specification being published as one of the outputs, (α, β) will be on the output list where α is the term address given by the hash root h of the proposition and β is the term address given by hashing h with the identifier of the current theory (and then tagging this with 33 to avoid accidental collision). If a proposition is imported by multiple different signatures being published, then the pair will be on the list multiple times. The information is obtained by calling signaspec_uses_props on appropriate preassets.

- output_doc_uses_objs takes an addr_preasset list (transaction outputs) and returns a list of pairs of term addresses. For each object imported as a parameter by a document being published as one of the outputs, (α, β) will be on the output list where α is the term address given by the hash root h of the term which was used to define the object and β is the term address given by hashing h with the type of the object and with the identifier of the current theory (and then tagging this with 32 to avoid accidental collision). If an object is imported by multiple different documents being published, then the pair will be on the list multiple times. The information is obtained by calling doc_uses_objs on appropriate preassets.

- output_doc_uses_props takes an addr_preasset list (transaction outputs) and returns a list of pairs of term addresses. For each proposition imported as a known by a document being published as one of the outputs, (α, β) will be on the output list where α is the term address given by the hash root h of the proposition and β is the term address given by hashing h with the identifier of the current theory (and then tagging this with 33 to avoid accidental collision). If a proposition is imported by multiple different document being published, then the pair will be on the list multiple times. The information is obtained by calling doc_uses_props on appropriate preassets.

- output_creates_objs takes an addr_preasset list (transaction outputs) and returns a list of triples (t, h, k) identifying objects defined in a document being published as one of the outputs. Here t is the (optional) hash value identifier of the theory in which the document lives, h is hash root of the term defining the object and k is the hash of the type of the object. (The term address of the pure object will be h and the term address of the object in the theory will be the hash of h with t, k and the tag 32.) If an object is created by multiple different documents being published, then

the triple will be on the list multiple times. The information is obtained by calling doc_creates_objs on appropriate preassets. If the pure term address for a created object is unowned, then it is new and must be given an owner (both as a pure object and as an object within the theory) with the same transaction publishing the document. If the pure term address for a created object is owned, but the term address within the theory is unowned, then the object is new for the theory and the term address within the theory must be given an owner (as an object) with the same transaction publishing the document.

- output_creates_props takes an addr_preasset list (transaction outputs) and returns a list of pairs (t, h) identifying propositions which are known as the result of a publication in the outputs. Usually, this will mean t is the (optional) hash value identifier of the theory in which the document lives and h is the hash root of a proposition proven in a document being published. Alternatively, a pair (t, h) can be included due to an axiom being assumed in a newly published theory specification. In this case, t is the hash value identifier of the theory derived from the new theory specification and h is the hash root of one of the propositions given as an axiom of the theory. Again, multiple publications may result in (t, h) being included multiple times. The function uses doc_creates_props. If the pure term address for a created proposition is unowned, then it is new and must be given an owner (both as a pure proposition and as a proposition within the theory) with the same transaction publishing the document. If the pure term address for a created proposition is owned, but the term address within the theory is unowned, then the proposition is new for the theory and the term address within the theory must be given an owner (as a proposition) with the same transaction publishing the document.

- output_creates_neg_props takes an addr_preasset list (transaction outputs) and returns a list of pairs (t, h) identifying propositions whose negations are proven in a document published in the outputs. Here this means there is a document being published in the theory identified by the (optional) hash value t and a proposition $\neg s$[7] is proven in the document and h is the hash root of s. The information is obtained by calling doc_creates_neg_props on appropriate preassets. There is no requirement to declare an owner for a created negated proposition. Negated propositions cannot be "used" in the sense that an object or proposition can be used. The only purpose for declaring ownership of a negated proposition is to collect a bounty.

- rights_out_obj takes an addr_preasset list (transaction outputs) and a term address α and sums the number of rights to use α as an object created by the outputs.

[7]Here $\neg s$ actually means literally negation $(\lambda_o x_0 \to \bot)$ applied to s or $s \to \bot$ where $\bot$ is $\forall_o x_0$.

- rights_out_prop takes an addr_preasset list (transaction outputs) and a term address α and sums the number of rights to use α as a proposition created by the outputs.
- count_obj_rights takes a list of assets and a term address α and sums the number of rights to use α as an object contained in the asset list.
- count_prop_rights takes a list of assets and a term address α and sums the number of rights to use α as a proposition contained in the asset list.
- count_rights_used takes list of pairs of term addresses and a term address α and counts the number of times α occurs as one of the pairs. (Technically it counts how many times α is at least one of the pairs, but in practice (α, α) will never occur.) The purpose is to determine how many times α is "used" by publications given in outputs of a transaction.
- obj_rights_mentioned takes an addr_preasset list (transaction outputs) and returns a list of term addresses. A term address is included in the output if object rights for it are being explicitly output (as RightsObj preassets) or if the object is being used in a document being published. (We do not count uses in signature specification publications since a signature specifications will only be allowed if rights are not required.)
- prop_rights_mentioned takes an addr_preasset list (transaction outputs) and returns a list of term addresses. A term address is included in the output if proposition rights for it are being explicitly output (as RightsProp preassets) or if the proposition is being used in a document being published. (We do not count uses in signature specification publications since a signature specifications will only be allowed if rights are not required.)
- rights_mentioned combines the results of obj_rights_mentioned and prop_rights_mentioned to give a list of all term addresses where rights are either being output or may be consumed to publish a document.
- units_sent_to_addr takes an address β and an addr_preasset list (transaction outputs) and sums the value of the currency units being sent to β. The purpose of this is to facilitate the purchase of rights by paying β.
- out_cost takes an addr_preasset list (transaction outputs) and sums the total cost of publishing the transaction. This includes the value of all currency and bounty assets created by the output as well as the burn cost required to publish theory specifications and signature specifications.

10.1.5 Asset Database

The module DbAsset implements a database for storing assets (see Chapter 8). At the moment, this uses the basic file storage implementation Dbbasic2. The function get_asset takes a hash value (an asset hash) and uses dbget to try to look up the asset from the database. If it is not found in the database, then the

asset is requested from network peers[8] and the exception GettingRemoteData is raised. The idea is that the next time get_asset is called the asset may have been put into the database after it was received from a peer.

Note: The keys to the assets in the database are the asset hash (given by hashasset), not the asset id (given by assetid).

10.1.6 Creation of Objects and Propositions

Let h be a theory identifier (an optional hash value), Δ be a document and α be a term address.

- We say (h, Δ) *creates the object* at α if there is a definition $\mathsf{DocDef}(\delta, s)$ in Δ where α is either the term address of the pure object s or of the object s of type δ in the theory with theory identifier h.
- We say (h, Δ) *creates the proposition* at α if there is a proof $\mathsf{DocPfOf}(s, \mathcal{D})$ in Δ where α is either the term address of the pure proposition s or of the proposition s in the theory with theory identifier h.
- We say (h, Δ) *creates the negated proposition* at α if there is a proof $\mathsf{DocPfOf}(s, \mathcal{D})$ in Δ and a proposition t where s is either $\neg t$ or $t \to \bot$[9] and α is the term address of the proposition t in the theory with theory identifier h.[10]

We extend these definitions from single documents to transaction outputs (which may publish several documents).

Let o be an addr_preasset list (transaction outputs) and α be a term address.

- We say o *creates the object* at α if there is some

  $$(\delta, (\omega, \mathsf{DocPublication}(\gamma, \nu, h, \Delta))) \in o$$

  where (h, Δ) creates the object at α.
- We say o *creates the proposition* at α if there is some

  $$(\delta, (\omega, \mathsf{DocPublication}(\gamma, \nu, h, \Delta))) \in o$$

  where (h, Δ) creates the proposition at α.
- We say o *creates the negated proposition* at α if there is some

  $$(\delta, (\omega, \mathsf{DocPublication}(\gamma, \nu, h, \Delta))) \in o$$

  where (h, Δ) creates the negated proposition at α.

[8] This is not currently implemented. Some earlier code to do this is commented out.

[9] Here $\bot$ is $\forall_o x_0$ and $\neg$ is $\lambda_o x_0 \to \bot$.

[10] We do not consider negated pure propositions. Negated propositions only need to be considered for collection of bounties by disproving a conjecture. Conjectures only make sense for propositions in a theory. (Note that every proposition is provable in an inconsistent theory, and so for every pure proposition there is some theory in which the proposition is provable.)

Ownership of an object, proposition or negated proposition will be originally justified by the creation of the object, proposition or negated proposition. We need a notion of *support* for this purpose. Ownership preassets are those of the form OwnsObj(β, p), OwnsProp(β, p) and OwnsNegProp. Let o be an addr_preasset list (transaction outputs) and α be a term address. We define when o supports an ownership preasset at α by considering the three kinds of preassets.

- We say o *supports* OwnsObj(β, p) at α if o creates the object at α.
- We say o *supports* OwnsProp(β, p) at α if o creates the proposition at α.
- We say o *supports* OwnsNegProp at α if o creates the negated proposition at α.

These notions will be used when we give the conditions for a ledger tree to support a transaction. In terms of the code, there is no single function checking if o support u at α. The functions output_creates_objs, output_creates_props, and output_creates_neg_props can be used to obtain term addresses which are created as objects, propositions and negated propositions. When we need to check for support in ctree_supports_tx_2 in the module ctre we will already have the values returned by output_creates_objs, output_creates_props, and output_creates_neg_props and will make use of these values at the time.

10.2 Transactions

The module tx defines a type tx for transactions, a type stx for signed transactions and functions for testing the validity of transactions and signed transactions. Here *validity* of a transaction refers only to properties that can be checked without reference to the state of the ledger. For properties that require the ledger state, we will speak of *support* for a transaction (see Chapter 11). There is a slight exception, however. We check validity of input signatures (check_tx_in_signatures) of a transaction relative to a list of assets being spent. These assets would need to be looked up in the ledger since the transaction only mentions the asset identifiers.

Note: Unit tests for the tx module are in `txunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

Note: The Coq module Transactions is intended to correspond to tx. The Coq types Tx and sTx correspond to the types tx and stx in the OCaml version. Readers can examine the formal properties proven in Transactions to have a better idea of what properties corresponding OCaml functions should satisfy. For more information, see [32].

The type tx of transactions is simply defined as a pair of an addr_assetid list (a list of pairs of addresses and hash value asset ids) and an addr_preasset list (a list of addresses associated with an obligation and a preasset).

```
type tx = addr_assetid list * addr_preasset list
```

In order to avoid needing to give multiple signatures corresponding to the same address, we allow transaction signatures to reference already given signatures. This is accomplished using the type gensignat_or_ref defined as follows:

```
type gensignat_or_ref = GenSignatReal of gensignat
                      | GenSignatRef of int
```

The idea is that we can have a list such at $[\sigma_1, \sigma_2, 0, 1]$ where σ_1 and σ_2 are of type gensignat giving real signatures and 0 and 1 are references to σ_2 and σ_1, respectively. The function getsig is used to determine the signature given a gensignat_or_ref and a list of signatures. In practice, the list of signatures given to getsig will be a list of the previous signatures. In the example $[\sigma_1, \sigma_2, 0, 1]$ the initial list of previous signatures is empty. In this case getsig is first called with the signature σ_1 returns σ_1 and the list $[\sigma_1]$. The next call to getsig would be with σ_2 and the list $[\sigma_1]$ returning σ_2 and the list $[\sigma_2; \sigma_1]$. The third call to getsig would be with the reference 0 and the list $[\sigma_2; \sigma_1]$ and return σ_2 (as element 0 on the list) and the unchanged list $[\sigma_2; \sigma_1]$. The fourth call to getsig would be with the reference 1 and the list $[\sigma_2; \sigma_1]$ and return σ_1 (as element 1 on the list) and the unchanged list $[\sigma_2; \sigma_1]$.

The type stx of signed transactions is a transaction associated with two lists of generalized signatures (gensignat) or references to other signatures.

```
type stx = tx * (gensignat_or_ref list * gensignat_or_ref list)
```

The first list gives "input" signatures and the second list gives the "output" signatures. The "input" signatures are required to spend or move the assets in the input. The "output" signatures are for the authors of publications. (Without these "output" signatures, a plagiarist could create his own transaction with someone elses publications and use his transaction to assign ownership of new objects and propositions.)

The serialization and deserialization functions are seo_tx, sei_tx, seo_txsigs, sei_txsigs, seo_stx and sei_stx.

We briefly describe the following exported functions:

- hashtx hashes a transaction, giving the identifier for the transaction (the *transaction id*). Note that this does not depend on signatures, and so transaction malleability is not an issue.
- tx_inputs is a projection function giving the input list of a transaction.
- tx_outputs is a projection function giving the output list of a transaction.
- no_dups is simply a helper function to ensure a list is duplicate free.[11]

[11] This is simply exported because the same function is required in the block module to ensure that no transaction is listed more than once in a block. As it has nothing to do with transactions, it should be moved to a more generic module imported by both tx and block.

- tx_inputs_valid takes a transaction input list and checks that there is at least one input and there are no duplicate inputs.

- tx_outputs_valid takes a transaction output list and checks that it is valid in that the following conditions hold:

    1. At most one owner (as an object or proposition) is declared for each term address.[12] The function checking this is tx_outputs_valid_one_owner.
    2. Each preasset is sent to an appropriate kind of address. Ownership preassets are sent to term addresses and publications and markers are sent to publication addresses.[13] The function checking this is tx_outputs_valid_addr_cats.

- tx_valid checks that both the inputs and outputs are valid in the sense above.

- tx_signatures_valid takes a block height b, an asset list and a signed transaction and checks that the input signatures and output signatures are valid. The work is partitioned in check_tx_in_signatures to check the input signatures and check_tx_out_signatures to check the output signatures.

    – check_tx_in_signatures ensures that for each input (except those spending markers and bounties) there is an appropriate input signature. Here there are two possibilities. There could be a signature permitting the spending of the asset (check_spend_obligation) or a signature permitting the movement of the asset (check_move_obligation). A signature permitting the spending of the asset is either a signature by the pay address in the obligation (assuming the appropriate block height has been reached) or by the address where the asset is held if there is no obligation.[14] A signature permitting the movement of the asset is by the address where the asset is held (assuming it is a pay address) and is only allowed if there an output with exactly the same obligation and preasset as the asset in question. Essentially this allows the "movement" of an asset out of an address to a new address.[15]

    – check_tx_out_signatures ensures the authors of all publications have signed the transaction. Note that the asset list is not required here.

[12] Note that it is legal for one term address to obtain an owner as an object and another owner as a proposition. In fact, this should be common for pure terms. For example, the term $\forall_o x_0 \to x_0$ (or $\top$) will have a hash root $h_\top$. This can be both defined and owned as an object as well as proven and owned as a proposition.

[13] This should probably be extended to ensure currency units and rights are only sent to pay addresses and bounties are only sent to term addresses.

[14] The obligation None defaults to being the address where the asset is held with no block height requirement.

[15] This could mitigate the effect of someone "spamming" someone else's address with unwanted and unowned assets.

- tx_signatures_valid_asof_blkh is like tx_signatures_valid but not given a block height. Instead it finds the minimum block height at which the signatures are valid and returns this block height (or None if there is no such block height).
- txout_update_ottree takes a transaction output list and uses it to update the current (possibly empty) ttree by including any new theories created by publishing theory specifications.
- txout_update_ostree takes a transaction output list and uses it to update the current (possibly empty) stree by including any new signatures created by publishing signature specifications.

10.2.1 Databases for Transactions and Signatures

The module DbTx implements a database for storing transactions and the module DbTxSignatures implements a database for storing transaction signatures (see Chapter 8). In both cases the key is the transaction id (the hash of the transaction, not including the signatures). To recover a value of type stx both of the values are required. At the moment, both of these use the basic file storage implementation Dbbasic2.

Chapter 11

Ledger Trees

The ctre module implements compact trees ("ctrees") and supporting functions. Compact trees are used to approximate the state of the ledger by recording what assets are held at what addresses. The ctregraft module implements a way to graft information onto a compact tree in order to form an approximation with more information.

Note: Unit tests for the ctre and ctregraft modules are in `ctreunittests.ml` in the `src/unittests` directory in the `testing` branch. These unit tests give a number of examples demonstrating how the functions described below should behave. The `testing` branch is, however, out of date with the code in the `dev` and `master` branches.

11.1 Compact Ledger Trees

We describe the main content of the module ctre.

Note: The largest part of the Coq formalization deals with verifying compact trees indeed represent ledgers as intended. The Coq module LedgerStates represents ledgers as functions describing the current state of the ledger (see the Coq type statefun). The Coq module MTrees uses a form of Merkle tree [19] to approximate such a statefun function. The Coq types hlist and nehlist defined in MTrees correspond to the types hlist and nehlist defined in ctre in the OCaml code. The dependent Coq type mtree n is a Merkle tree with height n, where the default case is $n = 162$ (since Qeditas addresses are determined by 162 bits). The Coq module CTrees uses the compact tree (essentially a Patricia tree) to represent the Merkle tree and hence approximate the ledger state. The dependent Coq type ctree n is a compact tree with height n. In the OCaml code the corresponding type is the simple type ctree. Several functions are defined by recursion on n in both the Coq and OCaml versions, even though in the OCaml version the requirement that the compact tree has height n is no longer enforced by the type system. Note that CTrees (mostly) corresponds to ctre, while LedgerStates and MTrees are only needed in the theory and (for the most

part) have no counterpart in the OCaml code. For more information, see [32].

11.1.1 Coin-age

We first consider some variables which affect the likelihood of coins to stake.[1] The current settings have been chosen after doing some simulations to determine a reasonable mixture between staking of coins in the initial distribution vs. staking of new coins issued through block rewards. (Initial simulations revealed that block rewards could easily dominate staking in the first few weeks unless their influence was dampened.)

Typically, unlocked currency assets age quadratically as $(1 + \lfloor \frac{a}{512} \rfloor)^2$ where a is the number of blocks since the asset became *mature*. This continues until a maximum age is reached. Users may commit currency assets to stake by locking them. Locked non-reward assets mature more quickly and, once mature, have their maximum age until it is close to the block height at which they will be unlocked, at which point they will be ineligible for staking. Rewards are necessarily locked. They age like unlocked currency assets until it is close to the block height at which they will be unlocked, at which point they will be ineligible for staking.

- maximum_age is the number of blocks after which unlocked coins stop aging. This is currently set to 2^{14}. With a 10 minute average block time, this means unlocked coins reach their maximum age after roughly 4 months. (Coins in the initial distribution are exceptional: they have birthday 0 and start with their maximum age.)

- maximum_age_sqr is $(1 + \lfloor \frac{a}{512} \rfloor)^2$ where a is maximum_age. This is the maximum factor that can be used to determine an asset's coin-age. Given current settings this is 33^2, i.e., 1089.

- reward_maturation indicates how old a reward must be before it can begin staking. This is currently set to 512.

- unlocked_maturation indicates how many blocks must pass before an unlocked asset can be used for staking. It is currently set to 512.

- locked_maturation indicates how many blocks must pass before a new locked currency asset can be used for staking. It is currently set to 8. This implies that 8 blocks after creating a locked currency asset, the currency asset can be used for staking with its maximum age, until it is close to being unlocked.[2]

- close_to_unlocked indicates the point at which locked assets can no longer be used for staking. It is currently set to 32, meaning that the locked asset

[1] Since this has nothing to do with compact trees, it should be moved to a more appropriate module.

[2] The intention here is to encourage stakers to commit to "locking" some of their coins only for use in staking.

cannot be used for staking if it is 32 blocks from being spendable (or even after it is spendable).

Rewards must be locked until a certain block height before they can be spent. In order to prevent rewards from dominating the early staking process,[3] this lock time is very long at first (16384 blocks, roughly 4 months), but reduces to 128 blocks (roughly 1 day) over the course of the first 114688 blocks (roughly 2 years). The function reward_locktime takes a given block height and returns the minimum number of blocks a reward must be locked. For the first 16384 blocks, the reward locktime is 16384 (the same as maximum_age). Every 16384 blocks, the reward locktime is halved until it reaches 128, where it remains indefinitely.

The function coinage computes the *coin-age* of a currency asset given the current block height. As described above the "age" ranges from 0 to 1089, with a quadratic increase each 512 blocks. (Specifically, the "age" progresses to be n^2 where n is incremented from 1 to 33 each 512 blocks and then remains at 33.) Assume v is the number of cants in the currency asset. If the birthday of the asset is 0, then it is part of the initial distribution and its coin-age is $1089v$. Other than coins in the initial distribution, there are three cases: unlocked currency assets, locked rewards and locked non-rewards. Unlocked currency asset and locked rewards mature after 512 blocks and then age quadratically as described above. In the case of locked rewards, the coin-age drops to 0 after block height $l-32$, where l is the lock given for the reward. Locked non-rewards mature after 8 blocks and then have coin-age $1089v$ until the coin-age drops to 0 after block $l-32$, where l is the lock given.

11.1.2 Approximating Asset Lists by Hlists

An asset list can be approximated by an *hlist* $\mathcal{H}$, a value of type hlist. The constructors for hlist are as follows:

- HHash(h) approximates a nonempty asset list with hash root h.[4]
- HNil approximates the empty asset list.
- HCons($a, \mathcal{H}$) where a is an asset and $\mathcal{H}$ is an hlist.
- HConsH($h, \mathcal{H}$) where h is an asset id (a hash value) and $\mathcal{H}$ is an hlist.

The idea is that an hlist explicitly lists a prefix of the assets (or references to the assets by their id) ending with a hash root summarizing the rest of the asset list. It is also possible for the hlist to list all the assets. Values of this type can be serialized and deserialized using seo_hlist and sei_hlist.

The type nehlist represents nonempty hlists and has three constructors:

[3] If rewards could be unlocked quickly, then they could be spent to create a locked non-reward asset. This locked non-reward asset would very quickly mature and begin staking with its maximum age.

[4] The hash root of an asset list does not seem to be explicitly defined in the code. However, it could be defined using ohashlist and hashasset and this seems to be the intended hash root.

- NehHash(h) corresponds to the hlist HHash(h).
- NehCons($a, \mathcal{H}$) corresponds to the hlist HCons($a, \mathcal{H}$).
- NehConsH($h, \mathcal{H}$) corresponds to the hlist HConsH($h, \mathcal{H}$).

Values of this type can be serialized and deserialized using seo_nehlist and sei_nehlist.

We briefly relevant functions.

- nehlist_hlist converts a nonempty hlist to an hlist.
- hlist_hashroot computes an optional hash root for an hlist, with None playing the role of the hash root for HNil.
- nehlist_hashroot computes a hash root for an nehlist, the same as the one given by hlist_hashroot.
- in_hlist and in_nehlist check if an asset is explicitly listed in the hlist or nehlist. Note that this will return false if the asset is not explicitly listed but is an asset on the part of the list being summarized by HHash or NehHash.
- print_hlist and print_hlist_to_buffer print hlists and are included for debugging purposes.

11.1.3 Compact Trees

The intention of a compact tree is to provide a binary tree approximation of a function from addresses (162-bit sequences) to hlists, where the hlists approximate the assets held at the addresses. In general, functions on compact trees will be defined by recursion and so we usually need to consider compact trees at level n (corresponding to functions from n-bit sequences to hlists). A 0 bit corresponds to the left child while a 1 bit corresponds to the right child.

The vast majority of the leaves will be empty, and so compact trees only store the nonempty parts explicitly. The empty compact tree (with no assets stored at leaves) can be thought of as represented by None, and the type ctree option is used when the empty compact tree should be considered.

Compact trees $\mathcal{C}$ are values of type ctree, which is defined by the following constructors.

- CLeaf($\overline{b}, \mathcal{H}$) is a compact tree with a single nonempty leaf at the location determined by the bit sequence (list of booleans) $\overline{b}$ and containing the nonempty hlist $\mathcal{H}$.
- CHash(h) is a compact tree with a hash h. This approximates every compact tree with hash root h.
- CLeft($\mathcal{C}$) is the compact tree with $\mathcal{C}$ as its left child and the empty tree as its right child.

- CRight($\mathcal{C}$) is the compact tree with the empty tree as its left child and $\mathcal{C}$ as its right child.

- CBin($\mathcal{C}_0, \mathcal{C}_1$) is the compact tree with two nonempty children: $\mathcal{C}_0$ on the left and $\mathcal{C}_1$ on the right.

Values of type ctree can be serialized and deserialized using seo_ctree and sei_ctree.

The following functions are important:

- ctree_hashroot computes a hash root for the compact tree. Many different compact trees will give the same hash roots, however they will always approximate the same ledger state. For example, they can differ in where they include CHash nodes, as well has the level of detail included in hlists at the leaves.

- octree_hashroot takes an optional compact tree and returns an optional hash value. It simply returns None for the empty compact tree None and returns the result of ctree_hashroot otherwise.

- ctree_lookup_asset takes an asset id (a hash value), a ctree and a bit sequence. It traverses to leaf in the ctree following the bit sequence. It then tries to look up the asset with the asset id in the nonempty hlist at the leaf. If the leaf is empty or the asset is not found, then None is returned. Otherwise, the asset is returned.

- ctree_addr given an address and a compact tree, returns the leaf (as a compact tree) at that address. This leaf could either be None indicating no assets are held at the address, or it could be a nonempty hlist, approximating the assets held at the address. It also returns the depth, which should always be 162, so it can be ignored.

- octree_lub takes the "least upper bound" of two optional compact trees. These are assumed to be "compatible" in the sense that they must have the same hash root. This means either both will be empty (None) or both will be compact trees. The least upper bound of two empty compact trees is the empty compact tree. For nonempty compact trees, the recursive structure is followed, abreviations are expanded, and if one of the trees is a CHash node then the other tree is taken. The idea is that if some information is in at least one of the trees, then the information will be in the resulting tree.

- print_ctree and print_ctree_all print compact trees and are included for debugging purposes.

11.1.4 Elements

As noted above, the purpose of having CHash(h) nodes in compact trees and having HHash(h) nodes in hlists is to work with small approximations of larger

structures. We will need to save some of these small approximations as key-value pairs in a database (see Chapter 8). This, however, introduces a problem: many different ctrees will have hash root h and can be approximated by the compact tree $\mathsf{CHash}(h)$. If we wish to use h as the key, there must be a unique compact tree (at least up to the level of detail included) that corresponds to h. A similar situation occurs when considering hlists.

To resolve these problems, we introduce a notion of *elements* for hlists and compact trees.[5] A compact tree element will be defined so that there is at most one compact tree element with a given hash root. We consider the case of hlists (and nehlists) first, as this is simpler.

First, empty hlists obviously do not need to be stored. It is clear that if the hashroot of an hlist is None, then the hlist list is HNil. Hence we only need to consider cons pairs. We say an *hcons element* is a pair (h, k) where h is a hash value and k is an optional hash value. The idea is that the hlist (or nehlist) corresponds to a nonempty asset list where the first asset has asset id h and the rest of the asset list has hash root k (or None if there are no more assets on the list). The weakest approximation of this asset list as an hlist that retains the h and k is $\mathsf{HConsH}(h, \mathsf{HHash}(k))$ if k is not None and $\mathsf{HConsH}(h, \mathsf{HNil})$ if k is not None otherwise. As an nehlist, the corresponding approximations are $\mathsf{NehConsH}(h, \mathsf{HHash}(k))$ if k is not None and $\mathsf{NehConsH}(h, \mathsf{HNil})$ if k is not None.

The module DbHConsElt implements a database for storing hcons elements (see Chapter 8). At the moment, this uses the basic file storage implementation Dbbasic2.

The function save_hlist_elements takes an arbitrary hlist and splits it into elements, saving the corresponding assets in the DbAsset database and the corresponding hcons elements in the database DbHConsElt. This ensures that there is enough information to reconstruct the hlist from the hash root and the information in the database. The function save_nehlist_elements performs a similar function for values of type nehlist.

The function get_hcons_element takes a hash value and uses dbget to try to look up the hcons element from the database. If it is not found in the database, then the data is requested from network peers[6] and the exception GettingRemoteData is raised. The idea is that the next time get_hcons_element is called the hcons element may have been put into the database after it was received from a peer. The functions get_hlist_element and get_nehlist_element call get_hcons_element and, if the hcons element is found, returns the information packaged as a value of type hlist or nehlist.

Compact tree elements are defined to give full information for 9 levels, using $\mathsf{CHash}(h)$ at the ninth level (and using $\mathsf{NehHash}(h)$ for all leaves). We can define this more formally as follows:

- A compact tree is *elemental at level 0* if it is of the form $\mathsf{CHash}(h)$.

[5] The notion of an element was introduced and coded by Trent Russell in early 2016. This replaced earlier code which used "frames" and "abbrev" nodes in compact trees.

[6] This is not currently implemented. Some earlier code to do this is commented out.

- A compact tree is *elemental at level* $i+1$ if it is one of the following forms: $\mathsf{CLeaf}(\overline{b}, \mathsf{NehHash}(h))$, $\mathsf{CLeft}(c_0)$ where c_0 is elemental at level i, $\mathsf{CRight}(c_1)$ where c_1 is elemental at level i or $\mathsf{CBin}(c_0, c_1)$ where c_0 and c_1 is elemental at level i.

- A compact tree is an *element* if it is elemental at level 9.

The function ctree_element_p checks if a compact tree is an element and the auxiliary function ctree_element_a checks if a compact tree is elemental at level i.

The ctree_super_element_a checks if a compact tree has at least as much explicit information as a compact tree that is elemental at level i. Likewise, the ctree_super_element_p checks if a compact tree has at least as much explicit information as a compact tree element. Such compact trees can be used to extract an approximating element. The function super_element_to_element computes such an approximation using the function super_element_to_element_a that computes an approximation that is elemental at a given level.

The module DbCTreeElt implements a database for storing compact tree elements (see Chapter 8). At the moment, this uses the basic file storage implementation Dbbasic2.

The function save_ctree_elements takes a compact tree and factors it into elements which are saved into the database, returning the compact tree $\mathsf{CHash}(h)$ where h is the hash root of the compact tree. The function makes use of save_ctree_elements_a which constructs intermediate elemental compact trees and uses dbput to save the constructed elements into the database.

The function get_ctree_element takes a hash value and uses dbget to try to extract the corresponding compact tree element from the database. If it is not found, then it is requested from peers[7] and the exception GettingRemoteData is raised. The idea is that the next time get_ctree_element is called the compact tree element may have been put into the database after it was received from a peer.

11.1.5 Transactions

We now describe functions relating compact trees and transactions. One of the main concepts is that of *support*. In short, we say a compact tree supports a transaction if for each input there is a corresponding asset held at the given address and that a number of conditions hold. To be more precise, there are further dependencies. For example, some of these conditions depend on the block height (e.g., to ensure an old enough intention justifies a publication).[8] Also, we must check the correctness of publications which may require looking up a theory or signature. These dependencies are explicit in the function

[7] This is not currently implemented. Some earlier code to do this is commented out.

[8] Lock heights are not checked here, but instead are checked with the signatures of transactions. The reason is that lock heights prevent an asset from being spent, but do not prevent assets from being moved. The distinction between being spent and being moved is not relevant for support.

ctree_supports_tx which checks if a compact tree supports a transaction, given a block height, theory tree (ttree) and signature tree (stree). The conditions to be checked often make reference to the actual assets being spent (which are referred to in the transaction by their assetids). The function ctree_lookup_input_assets elaborates the inputs by looking up the assets. The exception NotSupported is raised if some condition required for a transaction to be supported fails, which can happen either because the assets being spent cannot be found in the compact tree or because of failure of some condition.

Let $\mathcal{C}$ be a compact tree. We say an asset a is *held at* α *in* $\mathcal{C}$ if there is a nonempty hlist $\mathcal{H}$ at leaf α in $\mathcal{C}$ and a is explicitly listed in $\mathcal{H}$ (see in_nehlist). Let ι be an addr_assetid list (a list of transaction inputs) and ι' be a list of pairs (α, a) of addresses and assets. We say ι' is an *elaboration of* ι *relative to* $\mathcal{C}$ if for each $(\alpha, h) \in \iota$ there is a pair $(\alpha, a) \in \iota'$ where a is held at α in $\mathcal{C}$ and a has assetid h.[9] The function ctree_lookup_input_assets computes an elaboration ι' given ι and $\mathcal{C}$, raising NotSupported if there is no asset a with assetid h held at α in $\mathcal{C}$ for some (α, h) in ι.

The function ctree_supports_tx simply calls ctree_lookup_input_assets to obtain ι' and then calls ctree_supports_tx_2 with this extra information. The function ctree_supports_tx_2 checks the conditions for $\mathcal{C}$ to support $\tau = (\iota, o)$ with elaborated input ι' relative to a block height b, a theory tree and a signature tree.[10] Support requires several conditions. We describe each condition as a single sentence followed by a longer description.[11]

1. All output addresses are supported. That is, for each $(\alpha, u) \in o$ there is no CHash node along the path to the leaf in $\mathcal{C}$ with position α. This is needed so that the new assets can be added to the leaves. For each $(\alpha, u) \in o$ there are two possibilities: either there are no assets currently held at α or there are assets represented by the nonempty hlist $\mathcal{H}$ at α. If there are no assets, the new nonempty hlist will only contain the new assets. If there are currently assets represented by $\mathcal{H}$, we only need to push the new assets onto the explicit prefix of $\mathcal{H}$. Note that this is possible even if $\mathcal{H}$ is only the hash root of the hlist.

2. If an object or a proposition is used in a signature specification, then it must be free to use. For the definition of "used" see output_signaspec_uses_objs and output_signaspec_uses_props. Recall that each signature specification is intended for a specific theory (possibly the empty theory). Each parameter in a signature specification will correspond to two term addresses: one for the pure object and one for the object in the theory. Both of these addresses must be owned as objects and the ownership assets must both give 0 as the price of a right to use

[9] Technically, in the implementations the lists ι and ι' will list assetids and assets in the same order.

[10] The function ctree_supports_tx_2 is also given a list of assets, but this is simply the second components of the pairs in ι'.

[11] Similar conditions can also be found in the Coq formalization as ctree_supports_tx in `CTrees.v`.

the object. Likewise each axiom in a signature specification will correspond to two term addresses: one for the pure proposition and one for the proposition in the theory. Both of these addresses must be owned as propositions and the ownership assets also must give 0 as the price of a right to use the proposition. The reason for this condition is to prevent someone from paying once for the right to use an object or a proposition and then publishing it in a signature which is then free for anyone to use. Of course, someone can purchase the ownership assets from the owners and then make the corresponding objects and propositions free to use.

3. IF RIGHTS ARE SPENT IN THE INPUT, THEN THEY MUST BE MENTIONED IN THE OUTPUT. If a preasset $\mathsf{RightsObj}(\beta, n)$ is being spent, then β must be "mentioned" (see obj_rights_mentioned) in the sense that there is either an output of the form $\mathsf{RightsObj}(\beta, m)$ or the object is used (as a parameter) in a document being published. If a preasset $\mathsf{RightsProp}(\beta, n)$ is being spent, then β must be "mentioned" (see prop_rights_mentioned) in the sense that there is either an output of the form $\mathsf{RightsProp}(\beta, m)$ or the proposition is used (as an axiom) in a document being published.

4. RIGHTS MUST BE BALANCED. Let α be a term address corresponding to an object [a proposition] used in a document. Ensure that all the assets held at α are explict (using hlist_full_approx) and look up the ownership asset for α as an object [a proposition]. (Such ownership assets are held at α.) If α has no owner, then the transaction is not supported. Let r_1 be the number of rights to α used as an object [a proposition]. That is, r_1 is the number of documents which import α as a parameter [an axiom]. This is computed using count_rights_used. Let r_2 be the number of rights to α as an object [a proposition] which are created in the output of the transaction. This is computed using rights_out_obj [rights_out_prop] and can be described as a simple sum:

$$\sum_{\mathsf{RightsObj}(\alpha,m)\text{ in }o} m \left\lfloor \sum_{\mathsf{RightsProp}(\alpha,m)\text{ in }o} m \right\rfloor$$

Let r_3 be the number of rights to α as an object [a propostion] which are spent in the input of the transaction. This is computed using count_obj_rights [count_prop_rights] and can be described as a simple sum:

$$\sum_{\mathsf{RightsObj}(\alpha,m)\text{ in }\iota'} m \left\lfloor \sum_{\mathsf{RightsProp}(\alpha,m)\text{ in }\iota'} m \right\rfloor .$$

The function ctree_rights_balanced is called with this information, returning a boolean indicating if the rights are *balanced*. Rights being balanced depends on the cost to purchase rights which is found in the ownership asset. If the cost to purchase rights is None (meaning rights cannot be

purchased), then rights are balanced if $r_1 + r_2 = r_3$.[12] If the cost to purchase rights is 0, then the rights are balanced.[13] Assume the ownership asset is of the form $\mathsf{OwnsObj}(\beta, p)$ [$\mathsf{OwnsProp}(\beta, p)$] where $p > 0$. That is, the cost to purchase rights is $p > 0$. In this case it is possible rights are being purchased by paying cants to β. Let r_4 be the sum of cants sent to β in the output o. The rights are balanced if $r_1 + r_2 = r_3 + r_4$.[14] The transaction is not supported if the rights are not balanced.

5. Publications are correct, new and were declared in advance by a sufficiently old intention (marker). Recall that there are three kinds of publications: theory specifications, signature specifications and documents. The corresponding preassets are $\mathsf{TheoryPublication}(\gamma, \nu, \tau)$, $\mathsf{SignaPublication}(\gamma, \nu, h, \Sigma)$ and $\mathsf{DocPublication}(\gamma, \nu, h, \Delta)$. Recall that ν is a nonce. There will be two publication addresses associated with the publication. One is determined simply by the contents (τ, (h, Σ) or (h, Δ)) and must be the α (in o) where the publication asset will be held. (This is checked in tx_outputs_valid_addr_cats.) The α must be empty or the transaction is not supported. (If α holds an asset, it implies the publication has already been published.) All the information (including γ and ν) can be hashed to obtain a publication address β, called the *marker address*. The author of the document was able to compute this publication address β before making the corresponding publication (included in the preasset) public without revealing information about the publication's contents. The author must publish a Marker asset to the address β a certain number of blocks before attempting to publish the transaction with the publication.[15] The number of blocks is intention_minage which is currently set to 144 (a day, assuming 10 minute block times). This Marker must be spent by the transaction (assuring it exists) and must be old enough, otherwise the transaction is not supported. Finally, the publication must be correct. Correctness is judged by check_theoryspec, check_signaspec or check_doc. In the cases of check_signaspec or check_doc, the appropriate theory (if nonempty) must be looked up in the current theory tree, and the current signature tree must be given so that signatures imported by the signature specification or document can be retrieved. Also, recall that check_signaspec and check_doc depend on arguments gvtp and gvkn where gvtp determines if an term is known to have a type in a theory and gvkn determines if a proposition is known to be provable in a theory. Now that

[12] In practice, $r_1 > 0$ since the object or proposition was used. Hence the only way this equation can hold is if $r_3 > 0$. This is possible if rights to use the object or proposition were purchased earlier at a time when rights were being sold.

[13] Note that this means anyone can create rights to use the object or proposition later.

[14] There is a corner case here. If the same pay address β was used in more than one ownership asset at term addresses α_1 and α_2, then someone could simultaneously create rights to use α_1 and α_2 by paying cants to β once in a single transaction. This can be avoided by always giving a unique pay address for distinct ownership assets. This uniqueness is not enforced.

[15] The purpose of this is to prevent plagiarism, as described when Marker was introduced in Chapter 10.

we have access to the compact tree $\mathcal{C}$, we are in a position to supply these arguments. In particular, gvtp computes the term address α' for the object in the given theory (which depends on the theory identifier, the hash root of the term in question and the hash of the type in question) and uses hlist_lookup_obj_owner to determine if α' is known to have an owner as an object. Similarly, gvkn computes the term address α' for the proposition in the given theory (which depends on the theory identifier and the hash root of the term in question) and uses hlist_lookup_prop_owner to determine if α' is known to have an owner as a proposition. Since term addresses can only be given owners as objects [propositions] when they are published in a document, the type correctness [provability] is guaranteed by ownership.

6. If a marker asset is being spent in the input, then there must be a corresponding publication in the output. Suppose $(\beta, a) \in \iota'$ where the preasset of a is Marker. There must be some TheoryPublication(γ, ν, τ), SignaPublication(γ, ν, h, Σ) or DocPublication(γ, ν, h, Δ) in o for which β is the marker address.

7. If an ownership asset is spent in the input, then it must be included as an output. That is, once a term address has an owner, it will always have an owner. This is necessary since ownership is used to determine which objects have certain types and which propositions have been proven (in both cases relative to a theory). The ownership asset may be spent and recreated for a number of reasons. It can be done to change the pay address or the purchase price of rights. Such a change could correspond to the ownership being sold from one party to another, where the payment for the sale is part of the same (atomic) transaction. Another reason would be to collect a bounty. Ownership of propositions and negated propositions are spent (and recreated) as part of a transaction which collect bounties. (Collecting bounties is the only reason for ownership of negated propositions.) Note that the signature to spend an ownership asset is the pay address in the obligation, not the pay address for the purchase of rights. The pay address in the obligation indicates the "owner" of the term address.[16]

8. New ownership preassets in the transaction outputs must have an explicit (non-reward) obligation and must be created or transferred. If an ownership output is being created, it must be supported by the transaction outputs. Suppose $(\alpha, (\omega, u)) \in o$ where u is either OwnsObj(β, p), OwnsProp(β, p) or OwnsNegProp. The obligation ω must not be None and must indicate it is not a reward. There must either be an ownership asset at α listed in ι' (so ownership is being transferred) or no ownership asset is held at α in $\mathcal{C}$ (so the ownership is being created). If the ownership asset is being created, then o must support u at α (see Section 10.1.6).

[16] Ownership assets always have nontrivial obligations.

9. New objects and propositions must be given ownership by the transaction publishing the document. There are three different cases. (The notions of "creates" used here were defined in see Section 10.1.6.) If o creates an object at term address α and there is no OwnsObj asset held at α in $\mathcal{C}$, then there must be some $(\alpha, (\omega, \mathsf{OwnsObj}(\beta, p))) \in o$. If o creates a proposition at term address α and there is no OwnsProp asset held at α in $\mathcal{C}$, then there must be some $(\alpha, (\omega, \mathsf{OwnsProp}(\beta, p))) \in o$. (There is no requirement that created negated propositions must be given ownership.)

10. Bounties can only be collected by the owners of propositions or negated propositions. Suppose $(\alpha, a) \in \iota'$ and the preasset of a is $\mathsf{Bounty}(v)$. There must also be some $(\alpha, a') \in \iota'$ (with the same α) where the preasset of a' is either $\mathsf{OwnsProp}(\beta, p)$ (for some β and p) or OwnsNegProp. The fact that ownership asset is being spent means that the "owner" (as given in the obligation of a') must have signed the transaction spending the bounty. Note that by a condition above, the ownership asset must be recreated in the output o. The idea is that an owner of the proposition or negated proposition collects the bounty by a trivial transfer of the ownership asset.

An attentive reader will note that none of these conditions require the currency units consumed in the input to be at least as great as the currency units created in the outputs (plus those required to be burned to publish theories and signatures). This is not required for support, and will not be true for coinstake transactions (which receive a reward).

We summarize the descriptions of these three main functions discussed above as follows:

- ctree_lookup_input_assets takes a compact tree and an addr_assetid list (a list of transaction inputs) and uses ctree_lookup_asset to look up the assets corresponding to the assetids, returning the resulting list of pairs of addresses and assets. If one of the assetids cannot be found in the compact tree, the exception NotSupported is raised.

- ctree_supports_tx checks if a compact tree supports a transaction. The function also requires an optional ttree (with all the currently known theories) an optional stree (with all the currently known signatures) and the current block height. If the transaction is not supported, the exception NotSupported is raised. If the transaction is supported, the difference between the currency units output or burned and the currency units input is returned. If this value is negative, then it corresponds to a fee. If the value is positive, then it corresponds to a reward.

- ctree_supports_tx_2 is the same as ctree_supports_tx except it also receives two extra inputs: a list of the input addresses associated with their assets and a list of those assets.

If a compact tree supports a transaction or list of transactions, typically some small approximation of the compact tree also provides the support. We next describe functions to construct such small approximations.

- full_needed takes a addr_preasset list (a list of transaction outputs) and returns a list of bit sequences (addresses represented as boolean lists) indicating which leaves need to have their full list of assets explicit in order to check if the transaction with these outputs is supported.
- get_tx_supporting_octree takes a transaction and (optional) compact tree and returns an approximation of the (optional) compact tree sufficient to support the transaction.
- get_txl_supporting_octree takes a list of transactions and (optional) compact tree and returns an approximation of the (optional) compact tree sufficient to support the transactions.

There are two functions which transform (optional) compact trees using transactions.

- tx_octree_trans takes a block height, transaction and compact tree and transforms the ctree by deleting assets consumed in the inputs and create the new assets in the output. (The block height is needed to give birthdays to the new assets.)
- txl_octree_trans transforms a compact tree according to a list of transactions, sequentially.

There are also four auxiliary functions exposed in the interface.

- strip_bitseq_true takes a list of pairs, the first component of which are bit sequences, and returns the list filtered to the ones with a true as the head of the list with this true removed. For example, the input

$$[((\mathsf{false} :: \overline{b_0}), x); ((\mathsf{true} :: \overline{b_1}), y)]$$

  would give the output

$$[(\overline{b_1}, y)].$$

- strip_bitseq_false takes a list of pairs, the first component of which are bit sequences, and returns the list filtered to the ones with a false as the head of the list with this false removed. For example, the input

$$[((\mathsf{false} :: \overline{b_0}), x); ((\mathsf{true} :: \overline{b_1}), y)]$$

  would give the output

$$[(\overline{b_0}, x)].$$

- strip_bitseq_true0 takes a list of bit sequences, and returns the list filtered to the ones with a true as the head of the list with this true removed.

- strip_bitseq_false0 takes a list of bit sequences, and returns the list filtered to the ones with a false as the head of the list with this false removed.

These are exposed because they are used in the ctregraft module.[17]

11.2 Grafting Trees

The module ctregraft has code for grafting subtrees onto a compact tree in order to form an approximation with more information. The purpose of this is so that a block header can have a compact tree small enough to check the details of the asset which staked the block, and the block delta can have a graft extending this compact tree to a larger compact tree with enough information to support all the transactions in the block.

Note: In the Coq formalization the Coq module CTreeGrafting corresponds to ctregrafting. For more information, see [32].

The type cgraft is a list of hash values associated with compact trees. The idea is simply to associate some hash roots with compact trees with these hash roots. As usual, the serialization and deserialization functions for this type are seo_cgraft and sei_cgraft.

There are four functions exposed by ctregraft.

- cgraft_valid checks if a graft is *valid*, meaning simply that each pair $(h, \mathcal{C})$ is such that the hash root of $\mathcal{C}$ is h.
- ctree_cgraft takes a graft $\mathcal{G}$ and a compact tree $\mathcal{C}$ and replaces each $\mathsf{CHash}(h)$ in $\mathcal{C}$ with $\mathcal{C}'$ where $(h, \mathcal{C}')$ is in $\mathcal{G}$.
- factor_tx_ctree_cgraft takes a transaction and a compact tree $\mathcal{C}$ and computes a pair $(\mathcal{C}', \mathcal{G})$ of a compact tree $\mathcal{C}'$ and a graft $\mathcal{G}$.[18] Here $\mathcal{C}'$ is an approximation of $\mathcal{C}$ and ctree_cgraft applied to $\mathcal{G}$ and $\mathcal{C}'$ yields $\mathcal{C}$.
- factor_inputs_ctree_cgraft takes an addr_assetid (a list of transaction inputs) and a compact tree $\mathcal{C}$ and computes a pair $(\mathcal{C}', \mathcal{G})$ of a compact tree $\mathcal{C}'$ and a graft $\mathcal{G}$. Here $\mathcal{C}'$ is an approximation of $\mathcal{C}$ and ctree_cgraft applied to $\mathcal{G}$ and $\mathcal{C}'$ yields $\mathcal{C}$. The purpose of this function is to factor the part of the compact tree needed for the block header from the part needed for the rest of the block.

[17] It might make more sense to combine the ctre and ctregraft modules so that these functions need not be exposed.

[18] This function is currently unused.

Chapter 12

Blocks and Block Chains

The module block contains code related to blocks and block chains. This includes code to check if a block header is valid (including verifying the properties of the staking asset in the ledger), whether a block is valid and if a block or block header is a valid successor to a block or block header. In order to verify these properties we need to know when an asset is allowed to stake a block. We also allow for the possibility of forfeiture of block rewards as a punishment for signing on two different short forks.

Note: Unit tests for the block module have not been written.

Note: In the Coq formalization the Coq module Blocks corresponds to block. For more information, see [32].

12.1 Stake Modifiers

A *stake modifier* is a 256 bit number. The type stakemod is defined as four 64-bit integers as a way of representing such a 256 bit number. The functions seo_stakemod and sei_stakemod serialize and deserialize stake modifiers.

At each block height there will be a current stake modifier and a future stake modifier. The current stake modifier determines who will be able to stake the next block. The future stake modifier influences the next 256 current stake modifiers.

The genesis current and future stake modifiers should be set in the variables genesiscurrentstakemod and genesisfuturestakemod. These will determine who will be able to stake the first 256 blocks and will influence who will be able to stake the next 256 blocks, so it is important that these genesis stake modifiers are chosen in a fair manner. The function set_genesis_stakemods sets genesiscurrentstakemod and genesisfuturestakemod by taking a 160-bit number (as a 40 character hex string), applying one round of `SHA256` to obtain the value for genesiscurrentstakemod and another round of `SHA256` to obtain the value for genesisfuturestakemod.[1]

[1]The plan was to choose some Bitcoin height in the future and when that height was

The following three functions operate on stake modifiers.

- stakemod_pushbit takes a bit (as a boolean) and a stake modifier, shifts the 256-bit stake modifier (dropping the most significant bit) and using the new bit as the new least significant bit.
- stakemod_lastbit takes a stake modifier and returns its most significant bit (as a boolean).
- stakemod_firstbit takes a stake modifier and returns its least significant bit (as a boolean).

The current stake modifier changes from one block height to the next by taking the last bit of the future stake modifier and pushing this bit onto the current stake modifier. The future stake modifier changes from one block height to the next by pushing a new bit (either 0 or 1) onto the current future stake modifier. This implies those who stake blocks influence what will be the current stake modifiers, but this influence is limited. If one staker staked 50% of blocks, the staker would choose approximately 128 bits of the 256 stake modifiers in the future. The hope is that this influence is not enough to significantly improve their chances in the future, as each bit not chosen by the staker also has a large influence on who will be able to stake.

The function hitval performs one round of SHA256 on the least significant 32-bits of a 64-bit integer (intended to be the current time), a hash value (intended to be the asset id of the asset to stake) and a stake modifier (intended to be the current stake modifier). It returns the result as 256-bit number called the *hit value*.

12.2 Targets

A value of type targetinfo is a triple consisting of the current stake modifier, the future stake modifier and the current target (represented by a big_int). The target info used to determine if an asset is allowed to stake the next block. In particular, an asset can stake the hit value is less than the current target times the coinage of the staked asset (or, the coinage times 1.25 if proof of storage is used).

We have described above how the current and future stake modifiers change at each block height. The current target should also change in order to target an average 10 minute block. The function retarget defines how the target changes after each block.

- genesistarget is set to the initial target used for the genesis block.[2]

reached to obtain the 160-bit seed number from the last 20 bytes of the hash of the Bitcoin block header at that height.

[2]It is currently set to 2^{205}, but this should be reevaluated after a test run and before the launch of Qeditas.

- max_target is set to the maximum value for the target (i.e., the minimum difficulty). It is currently set to 2^{220}.

- retarget takes a target τ and a number of seconds Δ and returns a new target. The intention is that the given target is the current target and the number of seconds is the number of seconds between the current block and the previous block. The value is the minimum of either max_target or $\frac{\tau(9000+\Delta)}{9600}$. In particular, the value returned is never more than the value of max_target and remains τ if Δ is 600.

12.3 Proof of Storage

The consensus system for Qeditas is primarily proof-of-stake, but also includes a proof-of-storage component. A node can use evidence that it is storing some part of a term or document to increase the weight of its stake by 25%. The evidence is a value of type postor, defined by two constructors:

- PostorTrm(h, s, α, k) is evidence of storage of part of a term of a type in a theory at a term address. The optional hash value h identifies a theory, s is a term, α is a type and k is a hash value. Here s should have type α in the theory identified by h. The way this typing constraint is ensured is by checking that the term address correspond to the object s in theory h has an owner as an object. This ownership asset should have assed id k. The term s is intended to be minimal: all except exactly one left of the tree representing s should be an abbreviation (i.e., TmH of hash roots). (This minimality condition is checked by check_postor_tm_r.)

- PostorDoc$(\gamma, \nu, h, \Delta, k)$ is evidence of storage of part of a document at a publication address. Here γ is a pay address, ν is a hash value (nonce), h is an optional hash value (identifying a theory), Δ is a partial document (of type pdoc) and h is a hash value. The intention is that h is the asset id for an asset with preasset DocPublication$(\gamma, \nu, h, \Delta')$ held and the publication address determined by hashing γ, ν, h and Δ'. Here Δ' is a document with the same hash root as the partial document Δ. The partial document Δ should be minimal: with exactly one document item containing more than hashes and with that one document item only containing one explicit leaf, with others abbreviated by hash roots. (This minimality condition is checked by check_postor_pdoc_r.)

Values of type postor can be serialized and deserialized using seo_postor and sei_postor.

The exception InappropriatePostor is raised if a value of type postor is not an appropriate proof of storage because the term or partial document is not minimal.

- incrstake multiplies the number of cants being staked by 1.25. This is the adjusted stake used when proof of storage is included.

- check_postor_tm_r checks the minimality condition for a term, returning the hash of the unique important leaf upon success.
- check_postor_tm checks if PostorTrm(h, s, γ, k) can be used to increase the chances of staking. Let α be the (p2pkh) address where the asset to be staked is held. Let β be the term address for the object s of type γ in the theory identified by h. Let h' be the hash of the unique exposed leaf given by check_postor_tm_r. Let h'' be the result of hashing the pair of β and h'. Let h''' be the result of hashing α with h''. There are two conditions:
  1. A certain 16 bits of h''' are all 0. (This means that given a stake address α, only one in every 65536 items of the form PostorTrm(h, s, γ, k) can possibly ever be used to help α stake, independent of targets and stake modifiers.
  2. The hit value of h'' is less than the target times the adjusted stake.
- check_postor_pdoc_r checks the minimality condition for a partial document, returning the hash of the unique important leaf upon success.
- check_postor_pdoc checks if PostorDoc$(\gamma, \nu, h, \Delta, k)$ can be used to increase the chances of staking. Let α be the (p2pkh) address where the asset to be staked is held. Let β be the publication address for the corresponding document asset. Let h' be the hash of the unique exposed leaf given by check_postor_pdoc_r. Let h'' be the result of hashing the pair of β and h'. Let h''' be the result of hashing α with h''.
  1. A certain 16 bits of h''' are all 0. (This means that given a stake address α, only one in every 65536 items of the form PostorDoc$(\gamma, \nu, h, \Delta, k)$ can possibly ever be used to help α stake, independent of targets and stake modifiers.
  2. The hit value of h'' is less than the target times the adjusted stake.

12.4 Hits and Cumulative Stake

We now describe two functions for checking if an asset (optionally with proof of storage) is allowed to stake. This is sometimes informally referred to as "checking for a hit." A third function check_hit is deferred until we discuss block headers.

- check_hit_b is an auxiliary function which does most of the work to check if an currency asset can stake a block. It is given the block height, the birthday of the asset, the obligation of the asset, the number of cants v in the currency asset, the current stake modifier, the current target, the current timestamp, the asset id of the asset to stake, the p2pkh address holding the stake address[3] and an optional proof of storage. If no proof

[3]Note that the obligation of the stake address may mean that a different person can spend the staking asset than the holder who can stake the asset. This could be used to, for example, "loan" assets to someone else to stake.

of storage is given, the asset can stake if its hit value (relative to the time stamp and current stake modifier) is less than the product of the target and the coinage (as computed by coinage) of the asset. Suppose a proof of storage is given. In this case, we consider an adjusted stake using $1.25v$ instead of v. The asset can stake if the hit value of the asset is less than the target times the coinage of the adjusted stake and the proof of storage can be used (as judged by check_postor_tm or check_postor_pdoc).

- check_hit_a is simply a wrapper function which takes the target info (of type targetinfo) and calls check_hit_b after extracting the current stake modifier and current target from the target info. Factoring the functions this way makes it clear that check_hit_b does not depend on the future stake modifier.

The best block chain will be the one with the most cumulative stake.[4] The cumulative stake is represented by a big_int. The function cumul_stake computes the new cumulative stake given the previous cumulative stake, the current target τ and the latest delta time (time between blocks) Δ. It computes this by adding the following (big integer) value to the previous cumulative stake:

$$\lfloor \frac{\mathsf{max\_target}}{\tau \Delta 2^{-20}} \rfloor$$

or adding 1 if this value is less than 1.

12.5 Block Headers

We now describe block headers. A block header is made up of two sets of information: the header data and the header signature. The data part is represented using the record type blockheaderdata while the signature part is represented using the record type blockheadersig. A block header (of type blockheader) is simply a pair of the data with the signature. The functions seo_blockheader and sei_blockheader serialize and deserialize block headers. There is a value fake_blockheader which can be used when some data structure needs a block header to be initialized.

The fields in the record type blockheaderdata are as follows:

- prevblockhash should contain the hash of the data in the previous block header (or None for the genesis block header).
- newtheoryroot should be the hash root of the current theory tree (optional ttree) after the block with this header has been processed. It will change if some transaction in the block publishes a theory specification.
- newsignaroot should be the hash root of the current signature tree (optional stree) after the block with this header has been processed. It will change if some transaction in the block publishes a signature specification.

[4] The intention is also to have rolling checkpoints to prevent long range attacks.

- newledgerroot should be the hash root of the current compact tree (ctree) after the block with this header has been processed. This will always change since the asset staked will be spent and there will be outputs to the coinstake transaction of the block.
- stakeaddr should be the p2pkh address where the asset being staked is held.
- stakeassetid should be the asset id of the asset being staked.
- stored is an optional proof of storage (postor) and will be None if proof of storage was not used to help stake this block.
- timestamp is a 64-bit integer time stamp and should correspond to the time the block was staked.
- deltatime is a 32-bit integer which should contain the difference between the time stamp of this block and the time stamp of the previous block. (For the genesis block header, this should simply be 600.)
- tinfo should be the target information (current stake modifier, future stake modifier and current target) for this block header.
- prevledger is an approximation of the compact tree before processing the block corresponding to this block header. This approximation must contain the asset being staked and, if proof of storage is included, the relevant object ownership asset or document asset.

The fields in the record type blockheadersig are as follows:

- blocksignat is a cryptographic signature of type signat. This should be a signature of a hash of the data in the block header. Unless an endorsement is used, the signature should be by the private key corresponding to the stake address. If an endorsement is used, the signature should be by the private key
- blocksignatrecid is an integer which should be between 0 and 3. It is included to help recover the public key for the address (either stake or endorsed) from the signature (see the function `recover_key` in the module signat).
- blocksignatfcomp is a boolean indicating if the address (either stake or endorsed) corresponds to the compressed or uncompressed public key.
- blocksignatendorsement is an optional endorsement. If None, then signature corresponds to the stake address. Suppose it is (β, r, b, σ) where β is p2pkh address (the endorsed address), r is an integer $(0 \leq r \leq 3)$, b is a boolean and σ is a cryptographic signature. Here σ should be a signature of the Bitcoin message "`endorse` β" where β is the endorsed address (as a Qeditas address in base58 format). The signature σ should be by the

private key corresponding to the address α and r and b are used to recover the public key.

The following functions operate on block headers:

- blockheader_stakeasset takes block header data (blockheaderdata) and tries to return the staked asset by looking it up as stakeid at location stakeaddr in the compact tree prevledger. This can fail in two ways. First, it could be that the staked asset is not found, in which case an exception HeaderNoStakedAsset is raised. Second, it could be that prevledger includes more information than is necessary to give the staked asset, in which case an exception HeaderStakedAssetNotMin is raised.[5]

- hash_blockheaderdata hashes the data in the block header. This is to determine the hash to be signed in the signature part as well as the hash to be used in the previousblockhash field of the next block header.

- check_hit takes block header data (blockheaderdata) and checks if the given staked asset is allowed to create the block. It simply calles check_hit_a after extracting the target info (tinfo), time stamp (timestamp), stake asset id (stakeassetid), address where the staked asset is held (stakeaddr) and the optional proof of storage (stored) from given block header data.

- valid_blockheader determines if a block header is a valid block at the current height. In order to check if the block is valid the staked asset must be retrieved from the previous ledger. The staked asset must be a currency asset worth v cants. The auxiliary function valid_blockheader_a is called with the extra information given by this asset which in turn calls two (exported) functions: valid_blockheader_signat and valid_blockheader_allbutsignat. valid_blockheader_signat verifies the signature in the blockheader to be a valid signature (either directly or via endorsement) of the hash given by hash_blockheaderdata. valid_blockheader_allbutsignat checks the following conditions:

  1. The staked asset has the asset id declared in the header.
  2. The delta time is greater than 0.
  3. The staked asset is a "hit" for the current block height.
  4. If proof of storage is included, then the asset id given for the object ownership of the term or for the document is in the given approximation of the previous ledger.[6]

- blockheader_succ determines if a second block header is a valid successor to a first block header. The following conditions must be checked:

[5] The purpose of this condition is to prevent attackers from making unnecessarily large headers. The current implementation seems to be flawed, however, as it would not allow the relevant information from proof-of-storage to be included in prevledger.

[6] This probably no longer works if proof of storage is included, due to the minimality constraint on prevledger.

1. The second prevblockhash is the hash of the data in the first given block header.
2. The second timestamp is the sum of the first timestamp and the second deltatime.
3. The current stake modifier given in the second tinfo is the result of pushing the last bit of the future stake modifier of the first tinfo onto the current stake modifier of the first tinfo.
4. The future stake modifier given in the second tinfo is the result of pushing a 0 or a 1 onto the future stake modifier of the first tinfo.
5. The target given in the second tinfo is the result of retargeting using the target given in the first tinfo and the first deltatime.

12.6 Proof of Forfeiture

Proof of forfeiture is optional data proving a staker signed on two recent chain forks within 6 blocks. When such a proof is supplied by a staker of a block, the new staker can take recent coinstake rewards from the double signing staker. Such a proof is a value of type poforfeit and consists of a 6-tuple

$$(b_1, b_2, \overline{c_1}, \overline{c_2}, d, \overline{h}).$$

Here b_1 and b_2 are block headers which should contain different data but both be signed by the same stake address. The values $\overline{c_1}$ and $\overline{c_2}$ are lists of block header data each of which should have length at most 5. Finally, v is the number of cants being forfeited and $\overline{h}$ is a list of hash values (asset ids of the rewards being forfeited).

The function check_poforfeit verifies if the given value of type poforfeit can be used to support forfeiture of rewards. It first verifies that the data in b_1 and b_2 are different (by ensuring their hashes are different) and are staked using assets at the same stake address α. It also verifies that $\overline{c_1}$ and $\overline{c_2}$ have no more than 5 elements. It then verifies the signatures for b_1 and b_2. It calls check_bhl on $\overline{c_1}$ and $\overline{c_2}$ to ensure that each forms a (reverse) chain connecting b_1 and b_2 to some previous block hashes k_1 and k_2, and then checks that $k_1 = k_2$. This implies b_1 and b_2 are signed block headers forking from a common block (with hash k_1). (The function check_bhl also ensures that the hash of b_2 does not occur in $\overline{c_1}$ as this would mean the second chain is a subchain for the first, rather than a fork. Likewise it ensures the hash of b_1 does not occur in $\overline{c_2}$.) Finally it calls check_poforfeit_a which looks up assets by the asset ids listed in $\overline{h}$ and verifies that each is a reward less than 6 blocks old which was paid to address α and that the sum of these rewards is v cants.

12.7 Blocks

A *block* consists of a block header and a block delta. The block delta (implemented as the record type blockdelta) contains information about how to

transform the previous ledger (compact tree) into the next ledger (compact tree). In particular, the stake output is given (which completes the coinstake transaction) and all other transactions in the block are given. In addition, an optional proof of forfeiture is given which may effectively increase the rewards given to the staker of the block. In order to transform the previous ledger, one will generally need to graft more information about the previous ledger than was given in the header. This graft is also given.

The blockdelta record type consists of four fields:

- stakeoutput is the output to the coinstake transaction.
- forfeiture is an optional proof that a recent staker signed on a recent fork, thus justifying forfeiture of that staker's recent rewards.
- prevledgergraft is a graft providing the extra information needed by the output of the coinstake transaction, the other transactions in the block and optionally the data in the forfeiture field.
- blockdelta_stxl is a list of signed transactions, the transactions in the block.

The functions seo_blockdelta and sei_blockdelta serialize and deserialize block deltas.

The type block is the product of blockheader and blockdelta. The functions seo_block and sei_block serialize and deserialize blocks.

- coinstake builds the coinstake transaction by using the staked asset possibly combined with forfeited rewards as the input and taking stakeoutput from the block delta for the output.
- ctree_of_block returns the compact tree of a block (approximating the ledger state before processing the block) by taking prevledger from the block header data and grafting on prevledgergraft from the block delta. We call this the *compact tree of a block*.
- tx_of_block combines all the transactions in the block (including the coinstake) into one large transaction combining all the inputs and all the outputs. This is used to check validity of blocks.
- txl_of_block returns a list of all (unsigned) transactions in the block, including the coinstake transaction and the underlying transactions listed in blockdelta_stxl of the block delta.
- rewfn returns the number of cants of the reward at the current block height. The reward schedule is the same as Bitcoins (except for the amount of precision), except with the assumption that the first 350000 blocks have already passed (since this was the block height for the snapshot). We begin counting with a block height of 1. From blocks 1 to 70000, the block reward is 25 fraenks (2.5 trillion cants). After this the reward halves every 210000 blocks. Since the initial distribution contained (slightly less than) 14 million fraenks, this leads to cap of 21 million fraenks.

- valid_block checks if a block is valid at the given height. It does this by looking up the staked asset and passing the information to valid_block_a which checks the following conditions:

  1. The header must be valid.
  2. The transaction outputs in stakeoutput must be valid (as judged by tx_outputs_valid).
  3. If the staked asset has an explicit obligation, then ensure the first output on stakeoutput is of a preasset with the same amount of cants and the same obligation sent to the stake address.[7]
  4. All outputs in stakeoutput except possibly the first is explicitly must be marked as a reward and have a lock in the obligation at least as long as the value given by reward_locktime. Furthermore, all the outputs must be sent to the stake address. If the first output in stakeoutput is not marked as a reward, then it must also be sent to the stake address, must be a Currency asset with the same number of cants as the staked asset and must have the same obligation (possibly the default None obligation) as the staked asset.
  5. The compact tree of the block must support the coinstake transaction and it must have a reward at least[8] as high as the value given by rewfn.
  6. There are no duplicate transactions listed in blockdelta_stxl.
  7. The graft in prevledgergraft is valid.
  8. Each transaction in blockdelta_stxl has valid signatures, is valid and is supported by the compact tree of the block. Furthermore, none of these outputs are marked as rewards, none of these transactions spend the asset being staked. Finally, each transaction consumes at least as many cants as required.
  9. No two transactions in blockdelta_stxl spend the same input.
  10. No two transactions in blockdelta_stxl create ownership as an object (resp., as a proposition) at the same term address.
  11. If a transaction in blockdelta_stxl creates ownership as an object (resp., as a proposition) at a term address, then the output of the coinstake transaction does not create the same kind of ownership at the term address.[9]
  12. If proof of forfeiture is given, then check it is valid and remember the number of cants being forfeited.

[7] This is to support "loaning" assets for staking.

[8] This is to allow for collection of fees and of forfeiture of recent awards. The fact that the output is not too high is guaranteed later.

[9] It would make sense to simply disallow creation of non-currency assets in the coinstake transaction, but this is not currently the case.

13. Let $\mathcal{C}$ be the result of transforming the compact tree of the block (ctree_of_block) using the transactions of the block (txl_of_block). The hash root of $\mathcal{C}$ must be newledgerroot.
14. Let $\tau = (\iota, o)$ be the transaction of the block. The following must hold:
    - The cost of the outputs of τ (see out_cost) is equal to the sum of the assets being spent along with the reward (rewfn) and (possibly) the number of cants being forfeited.
    - The transformation of the current theory tree by o must have hash root newtheoryroot.
    - The transformation of the current signature tree by o must have hash root newsignatroot.

    Upon success, valid_block returns the transformed theory tree and the transformed signature tree. Upon failure, either None is returned or an exception is raised.

12.8 Databases for Block Information

There are three databases for blocks, all using the hash of the block header as the key. The module DbBlockHeader is a database for block headers (implemented using Dbbasic2keyiter) and the module DbBlockDelta is a database for block deltas (implemented using Dbbasic2).

12.9 Chains

There are additional types blockchain and blockheaderchain. These can be used to represent (nonempty) chains of blocks or block headers.[10] In each case the representation is as a pair where the first component should be the most recent block of block header and the second component is a list of the previous blocks or block headers in reverse order.

The variable genesisledgerroot gives the ledger root of the initial compact tree with the initial distribution. The value is (as of September 2016):

```
fc25150b4880e27235d4878637d32f0ffe2280e6
```

- blockchain_headers converts a block chain into block header chain by dropping the block deltas.
- ledgerroot_of_blockchain takes a block chain and returns the value of newledgerroot in the latest block header data.
- valid_blockchain checks if a block chain is valid at a given height. This requires checking the validity of each block and that each block header is

[10] It is not clear if this is explicitly needed.

a valid successor to the previous block header. It also requires keeping up with the theory tree and signature tree. In the case of the genesis block, the prevblockhash should be None, the prevledger should have hash root genesisledgerroot, the tinfo should be composed of the values in genesisccurrentstakemod, genesisfuturestakemod and genesistarget and the deltatime should be 600.[11]

- valid_blockheaderchain checks the validity of a block header chain. It is similar to valid_blockchain but only checks the headers are valid instead of the full blocks.

[11] Alternatively, one could set a "genesis timestamp" and enforce that the deltatime of the genesis block is the difference between the time stamp of the genesis block and the fixed genesis timestamp.

Chapter 13

Block Trees

The module blocktree (`blocktree.ml` and `blocktree.mli`) contains code related to keep up with the current tree of blocks and the current best block. Other various information is also included here.

13.1 Block Tree

The main data type in the module blocktree is the type blocktree with one constructor BlocktreeNode (described below). Each node can be thought of as between a block and possible successor blocks. Enough information is included in the node to determine if a new block (header) is a valid successor block (header). The root of the tree is the "genesis node" which contains information about the initial ledger tree, initial stake modifier and initial target. Qeditas has no "genesis block" in the traditional sense. The first block would be a valid child of the genesis node.

Nodes to the tree are added as headers are received (or staked) even if the corresponding block delta has not yet been received. In order to distinguish between these cases there is a type validationstatus with three constructors: Waiting (meaning the block delta is being requested from peers), ValidBlock (meaning a delta has been received and checked to be part of a valid block when combined with the header) or InvalidBlock (meaning there is no delta making the header part of a valid block, though it is not clear how this could be determined).

The single constructor BlocktreeNode of the type blocktree takes 13 arguments:[1]

- an optional blocktree giving the parent, which is None for the genesis node (and possibly for some checkpoint nodes).
- a reference to a list of hash values, giving a list of addresses which have staked in some child of this node up to six levels deep. This is to be used

[1] This probably should be a record type.

to identify double-staking which can be punished by proof-of-forfeiture.

- an optional hash of the previous block header, which is None for the genesis node.
- an optional root of the current theory tree (see ttree in Chapter 9).
- an optional root of the current signature tree (see stree in Chapter 9).
- the current ledger hash root.
- the current target information (including the current and future stake modifier).
- the current time stamp.
- the current cumulative stake.
- the current block height.
- a reference to the current validation status (of type validationstatus).
- a reference to a boolean indicating if the block above the node has been blacklisted, so children should not be considered valid.
- a reference to a list of children, which gets updated as successor blocks are found.

There are a variety of functions for obtaining the information above from a node in the tree: node_recent_stakers, node_prevblockhash, node_theoryroot, node_signaroot, node_ledgerroot, node_targetinfo, node_timestamp, node_cumulstk, node_blockheight, node_validationstatus and node_children_ref.

The function eq_node tests if two nodes are equal by simply checking if they have the save previous block hash. The usual equality operation cannot be used since the target is of type big_int whose values are (often) abstract.

The variable genesisblocktreenode is initialized to be the node with the genesis ledger root, genesis target information and genesis time stamp. A child of this node corresponds to a block at height 1.

The hash table blkheadernode associates hash values with their corresponding node in the block tree.

The functions process_new_header, process_new_header_a, process_new_header_aa, process_new_header_ab and process_new_header_b are used to check a header is valid and, if so, create a node for it and enter the node into the block tree. If the previous block hash in the header does not correspond to a node in the tree (according to blkheadernode), the new header is considered an orphan and put into the hash table orphanblkheaders. The parent header of an orphan block header are requested using find_and_send_requestdata, if possible. When a new header is processed, it may mean previously orphaned headers are no longer orphaned, at which point the headers should be removed from orphanblkheaders

and new nodes in the block tree should be added. (The handling of orphan blocks has not been tested.)

In addition, there is a list earlyblocktreenodes which contains headers which have been received but whose time stamps are in the future. These are intended to be delayed until the appropriate time arrives (although the code to handle them later does not seem to be written).

The hash table tovalidate is a set of hashes of headers where we are still waiting to validate the corresponding block delta.

13.2 Checkpoints

The checkpoint public key is given in this module by checkpointspubkeyx and checkpointspubkeyy. If the corresponding private key is given on the command line or in the configuration file, then checkpointsprivkeyk is set to the corresponding integer. (The key is for the uncompressed address, and this fact is hard-wired into the code.)

The value lastcheckpointnode is set to the most recent checkpoint, which is the genesis node by default.

The hash table checkpoints associates header hashes with pairs (h, σ) where h is a block height and σ is a signature of the hash by the checkpoint key.

13.3 Best Node

The current block chain is determined by finding the "best node" in the block tree, i.e., the node with the highest cumulative stake. We sometimes allow the best node to be awaiting validation and sometimes require the best node to be fully validated.

The variable bestnode is set to the current best node. This gets reset when a new block is staked, if a new header is received or if Qeditas gives up waiting for a node to be validated. In the last case, we may want to find the best validated node. The value of bestnode should never be updated directly, but instead should be updated by calling update_bestnode. Given a node n, update_bestnode sets bestnode to n, sets netblkh to the current block height (technically, the height of the next block to be found) and, if the private key for signing checkpoints is known, the block which is now 36 blocks deep is signed as a checkpoint, lastcheckpointnode is updated and the checkpoint is broadcast (as inventory) to all peers.

The function initblocktree creates a genesis block tree node and initializes the values genesisblocktreenode, lastcheckpointnode and bestnode accordingly. Then init_headers is called to traverse all known block headers from the block header database DbBlockHeader and create corresponding nodes in the block tree.

The function find_best_validated_block_from finds the descendent node with validation status ValidBlock (and which is not blacklisted) with the highest cumulative stake. The best node and cumulative stake found so far are ex-

tra arguments to compare against. The function find_best_validated_block calls find_best_validated_block_from starting from the current value of lastcheckpointnode and calls update_bestnode with the result.

The function print_best_node gives the hash of the last block header of the current best node (or endicates it is the genesis node if there is no previous header).

The function record_recent_staker climbs an indicated number of parents from a given node and inserts a given address into the recent stakers of the node. This information is then used by is_recent_staker to determine if an address has staked on some descendent of a node (up to some depth). The relevant depth is 6, but the functions are recursive, so the depth arguments i are generic.

13.4 Other Local Data

The hash table stxpool associates hash values with corresponding signed transactions and published_stx records which of these transactions have been published (broadcast to peers).

The hash table thytree associates hash values with the root of a corresponding theory tree ttree. The function lookup_thytree looks up the theory tree given an optional hash value, where the default value of None returns the empty theory tree None. For the moment, Qeditas has only been tested with empty theory trees.

Similarly, there is a hash table sigtree which associates hash values with the root of a corresponding signature tree stree. The function lookup_sigtree looks up the signature tree given an optional hash value, where the default value of None returns the empty signature tree None. Again, Qeditas has only been tested with empty signature trees.

13.5 Networking Code

Since the blocktree module has access to the current state of the block chain, a function send_inv is defined to collect the most recent inventory data (transactions, block headers and block deltas) to send to new peers. The networking code obtains access to the send_inv function via the reference send_inv_fn which is set to send_inv in blocktree.

The functions publish_stx and publish_block add transactions and blocks to the local database and broadcast them as new inventory to peers.

In addition, most of the implemented message handlers for the networking code (see Chapter 7) are given in the blocktree module by associating functions with message types in the hash table msgtype_handler. Here are the message types handled in blocktree:

- Inv: The payload of an inventory message should be a 32-bit integer n indicating how many inventory items there are followed by n triples (τ, h, k) where τ is a message type (as a byte, indicating the kind of object in the

inventory), h is a block height (as a 64-bit integer) and k is a hash value (indicating the identifier of the object in the inventory). The handler reads these triples and determines whether or not to request the corresponding objects. Headers will be requested if they are new and those to be requested are collect into a list to be requested in batches of at most 255 headers (by req_header_batches). Block deltas are requested if they are new and the corresponding header is waiting to be validated (as indicted by tovalidate). If a signed transaction is new, then the corresponding unsigned transaction is requested. If the transaction of a signed transaction is known but the signatures are not, the signatures are requested. A checkpoint is requested unless it is a known checkpoint or would be an ancestor of the current checkpoint. No other inventory is immediately requested. All inventory is inserted into the rinv field of the connection state in case we need to request something later.

- GetHeader: The payload contains a hash value. If the corresponding header has already been sent to the node (as recorded in sentinv), then it is not resent. Otherwise, the header is loaded from the database and sent to the peer in a Headers message. If the header is not in the database, the message is ignored.

- GetHeaders: The payload contains a byte n indicating the number of headers being requested, followed by n hash values. The corresponding headers which have not yet been sent are collected and send to the peer in a Headers message.

- Headers: The payload contains a byte n indicating the number of headers being sent, followed by n hash values and block headers. Each header must hash to the given hash value, otherwise this is logged and the connection is marked as banned by setting banned in the connstate to true. Each new header is checked to be valid and, if so, processed by process_new_header_ab (which assumes the given header is valid). If a received header is not valid, this is logged and the connection is marked as banned by setting banned in the connstate to true.

- GetBlockdelta: The payload contains a hash value identifying a block. If the corresponding block delta has not yet been sent to the peer, it is sent in a Blockdelta message.

- Blockdelta: The payload contains a hash value and a block delta. If the hash value is of a block for which the block delta is already known, then the handler stops (before deserializing the block delta). Otherwise, the node for the corresponding header is found in the block tree. If there is no such node in the block tree, the message is ignored. Assume there is such a node n. We check its validation status. The case we are interested in is when the validation status is Waiting but there is not yet a candidate blockdelta to validate. In this case we find the parent node p of n. If the parent node is already known to correspond to a valid block (or there is no parent node),

then the block delta is deserialized and the function validate_block_of_node is called to check that the new block is valid and update the validation status of the node. (Assuming the block is valid, validate_block_of_node also traverses children which have block deltas waiting to be validated and tries to validate these.) If the parent node has not yet been validated, then the block delta is deserialized and stored (along with the connection state of the peer) with the Waiting status. In principle it will be validated after its ancestors have been validated.

- GetTx: The payload is a hash value which should be a transaction id. Unless the transaction has already been sent to the peer, the (unsigned) transaction is looked up in stxpool or DbTx and sent as a Tx message.

- Tx: The payload is a hash value h followed by an unsigned transaction (of type tx). If h is the transaction id of a transaction in the DbTx database, we ignore it. Likewise, if the transaction was not explicitly requested from the peer, we ignore it. Otherwise, we deserialize the transaction and check that it hashes to h and put it into the DbTx database.

- GetTxSignatures: The payload is a hash value which should be a transaction id. Unless the transaction signatures have already been sent to the peer, the transaction signatures are looked up in stxpool or DbTxSignatures and sent as a TxSignatures message.

- TxSignatures: The payload is a hash value h and a collection of transaction signatures (see Chapter 10). If h is the transaction id of a transaction in the DbTxSignatures database, we ignore the message. Likewise, if the transaction signatures were not explicitly requested from the peer, we ignore the message. Otherwise, the signatures are deserialized and checked to be complete and valid (for some block height). Assuming the signatures are complete and valid, the signatures are saved in DbTxSignatures and added to the transaction in stxpool.

- GetCheckpoint: The payload is a hash value, which should be the hash of a block header which has been signed as a checkpoint. If the checkpoint exists and has not previously been sent to the peer, it is sent as a Checkpoint message.

- Checkpoint: The payload is a hash value h (identifying a block header), a block height i and a signature σ. The signature σ is checked to be a valid signature of h for the public key for checkpoints. Assuming it is valid, the pair (i, σ) is added to checkpoints. If the hash value h was previously blacklisted in DbBlacklist, the entry is removed from the blacklist. Assume there is a node n corresponding to h in the block tree. If the node n is marked as blacklisted, this is reset to false while all the siblings of n (if there are any) are marked as blacklisted. The node is set to have validation status Valid (independent of whether a block delta was ever received). If

there is no node corresponding to h in the block tree, a node is simply created (with no parent).[2]

13.6 Dumping the State

The function dumpblocktreestate writes a large amount of information about the current block tree state into a channel (presumably of an open text file). This is for debugging purposes.

[2]This could cause confusing with the genesis node in some parts of the code, and should probably be handled differently.

Chapter 14

Commands

The commands module (`commands.ml` and `commands.mli`) contains code for handling top level commands (from the command line interface).

The module commands is intended to support a variety of commands a user may need. At the moment it only supports limited wallet and transaction creation commands. Some state information is held in this module (although it likely should be moved elsewhere).

- walletkeys contains the private keys in the wallet. More specifically it is a list of values of the form $(k, b, (x, y), w, h, a)$ where k is the private key, b is a boolean indicating if it is for the compressed public key, (x, y) is the public key, w is the string base-58 WIF format, h is the 20-byte hash value corresponding to the p2pkh address and a is the string base-58 Qeditas p2pkh address.

- walletp2shs contains entries of the form $(h, a, \overline{b})$ where h is the 20-byte hash value of a p2sh address, a is the base-58 Qeditas p2sh address and $\overline{b}$ is the sequence of bytes giving the script corresponding to h. Note that this does not directly give a way of "signing" for the p2sh address.

- walletendorsements contains the endorsements in the wallet. In particular it is a list of values of the form $(\alpha, \beta, (x, y), b, \sigma)$ where α and β are pay addresses,[1] (x, y) is the public key for α, b is a boolean indicating if α is the address for the compressed public key and σ is a signature for a Bitcoin message of the form "endorse β" (or "testnet:endorse β" in the testnet) signed with the private key for α. The private key for β should be in walletkeys and this private key along with the endorsement means the wallet can sign for α.[2]

[1] Actually, in what is implemented we assume they are both p2pkh addresses. In principle endorsements involving p2sh addresses are supported by the code in sigant and script, but support has not been implemented in commands.

[2] The endorsement mechanism gives Bitcoin users a way to claim their part of the initial distribution without revealing their private keys.

- walletwatchaddrs contains addresses to "watch."
- stakingassets contains a list of assets which the node can stake. This changes as bestnode from blocktree changes.
- storagetrmassets is intended contain a list of assets at term addresses which the node can use as proof-of-storage to improve the chances of staking. Currently it is unused.
- storagedocassets is intended to contain a list of documents at publication assets which the node can use as proof-of-storage to improve the chances of staking. Currently it is unused.
- txpool is a hash table associating hash values (transaction ids) with signed transactions. This is loaded and saved to the file `txpool`. The intention is that this holds transactions which have been published but are not yet included in a block.

The following functions are for loading and saving the state in certain files.

- load_txpool sets txpool by loading the contents fo `txpool`.
- load_wallet sets walletkeys, walletp2shs, walletendorsements and walletwatchaddrs by loading the file `wallet`.
- save_wallet saves the current wallet contents (the values of walletkeys, walletp2shs, walletendorsements and walletwatchaddrs) in `wallet`.
- printassets prints the assets from the current ledger tree (the ledger tree with root in the current best node) at the addresses mentioned in the wallet. If some of the relevant parts of the ledger are missing and there are connections to peers, an attempt is made to request the relevant information from peers. If this is taking too long, it may give up and print partial information (meaning that balances appear smaller than they are). Calling printassets multiple times with sufficient peers should eventually fetch enough information about the ledger to give all assets and the full balance.
- printassets_in_ledger prints the assets in a ledger with an explicitly given ledger root at the addresses mentioned in the wallet.
- printctreeinfo prints a summary of information about the compact tree with the given hash root.
- printctreeelt prints the ctree element (complete information for exactly 9 levels) with the given hash root.
- printhconselt prints the hcons element with the given hash root, meaning it prints the asset id and optionally the root for the next hcons element.

- printasset prints the asset with the given asset hash. Note: the hash of the asset must be given, not the asset id. The keys to the assets in the database are the asset hash (given by hashasset), not the asset id (given by assetid).
- printtx prints the transaction with the given hash root.
- btctoqedaddr parses a Bitcoin address (base-58 representation) and prints the corresponding Qeditas address (base-58 representation).
- importprivkey imports a private key given in Qeditas WIF.
- importbtcprivkey imports a private key given in Bitcoin WIF.
- importendorsement imports an endorsement.
- importwatchaddr imports a Qeditas address to watch.
- importwatchbtcaddr imports a Bitcoin address in order to watch the corresponding Qeditas address.
- createsplitlocktx creates a transaction to split an asset into several assets with a given lock height.
- signtx signs a transaction.
- savetxtopool saves a transaction to the local transaction pool (without publishing it).
- sendtx publishes a transaction by sending it to peers.

Chapter 15

Staking Code

The staking code is part of the top level file `qeditas.ml`. A thread is created via the function stakingthread if the configuration variable staking is set to true. stakingthread is a loop which takes the current value of bestnode and tries to find an asset which can stake the next block within the next hour or two.

The hash table nextstakechances associates block header hash values with information about the next chance to stake on top of the corresponding block. The information is of type nextstakeinfo which has to constructors:

- NextStake$(i, \alpha, h, b, \omega, v, c)$ meaning the next chance to stake is at type i with the currency asset worth v cants with id h, birthday b and obligation ω held at (p2pkh) address α and that the resulting block will have cumulative stake c.

- NoStakeUpTo(i) meaning there is no chance to stake up to time i.

The entries in nextstakechances are computed by the function compute_staking_chances. compute_staking_chances takes a blocktree node n and two integers t_0 and t_1 giving the starting time and ending time and searches for the first $i \in [t_0, t_1]$ where some held currency asset can stake. First, it collects the assets held at p2pkh addresses where either a private key or an endorsement is in the wallet. These assets are put into stakingassets from the commands module. If compute_staking_chances finds an asset that can stake, the relevant information is put into nextstakechances. If none is found, NoStakeUpTo(t_1) is put into nextstakechances.

Assume there is an asset which can stake. If the next chance to stake is greater than 10 seconds away, the thread sleeps for a minute and checks again (in case the best node has changed in the past minute). If it is within 10 seconds of the time to stake, the block header and delta are constructed and published.[1]

[1] A final validity check is done before publishing the block. In principle all blocks constructed should be valid, but in practice sometimes invalid blocks are constructed. This is presumably due to bugs.

If no asset can stake within the next hour, the thread continues sleeping for 1 minute before rechecking the current state. Eventually either the best node changes or enough time has past that we call compute_staking_chances again for more information.

In some cases the exception StakingProblemPause is raised. If this happens, it is considered a problem caused by a bug. The staking thread deals with this by pausing the thread for an hour and then trying to stake again in the new state.

Chapter 16

Top Level Code

The file `qeditas.ml` is used to create an executable `qeditas`. This executable starts several threads for networking, staking (optionally) and finally for a console-style interface.

To understand what the executable `qeditas` does, see the code near the end of the file `qeditas.ml`. We describe the tasks `qeditas` executes briefly. The function initialize is called first and is described in Section 16.1. Next the function initnetwork is called, see Section 16.2. If the configuration variable staking is set to true, then a thread is created which calls the function stakingthread, see Chapter 15. Finally, a main loop begins for reading and processing commands from the console begins, see Section 16.3.

16.1 Initialization

The initialize function begins by checking the command line arguments for an option of the form `-datadir=...` which would override the default data directory (usually `.qeditas` in the user's home directory). It then reads the config file `qeditas.conf` in the data directory. This may override some default values in the config module. It then reads the other command line arguments which may again override some values in the config module. If the testnet configuration variable is true (which is mandatory at the moment), then the `testnet` subdirectory of the main data directory is used. This implies the testnet will have its own databases, its own wallet, and so on. The database directory is set and configured and a log file is opened.

If there is a `.lock` file in the data directory, then Qeditas exits to prevent two instances of Qeditas from using the same data directory. Otherwise, Qeditas creates a `.lock` file in the data directory and sets the variable exitfn to a function which calls saveknownpeers (to remember the peers discovered during the session) and remove the `.lock` file. At this point, the only way Qeditas should exit is via a call to exitfn.

The database is initialized by calling dbconfig with a db subdirectory of the

data directory and then calling dbinit on each database.

Next the log file is opened (via openlog).

The code then checks if either the seed variable in config has been set to a nonempty string. (It should be a 40 character hex string.) The current code in setconfig sets seed to

```
68324ba252550a4cb02b7279cf398b9994c0c39f
```

unless it is specifically set in the configuration file or on the command line. The value above is the last 20 bytes of the hash of Bitcoin block 378800, and was a value included only for testing purposes. The intention was to choose a Bitcoin block height roughly one week in the future when the time came for Qeditas to launch. The last 20 bytes of that block hash would be the value for seed. The purpose of this value is to initialize the current stake modifier and the future stake modifier (see set_genesis_stakemods). These stake modifiers affect which assets will stake within the first 512 blocks. In particular it affects the genesis block (at Qeditas block height). For the launch to be fair, these stake modifiers should not be predictable before launch. This goal could be accomplished in other ways. The possibility is left open in the future that seed is not set but that a checkpoint has been set so that a node can begin following the block chain from that checkpoint. (The full history is not required. Each ledger tree contains the full information required to continue.)

The function initblocktree from the blocktree module is called. In particular, this processes all currently known headers in the database to build a block tree and determine the current best node (and hence block chain).

If the testnet configuration variable is set, then the difficulty is decreased significantly (setting genesistarget to 2^{200} so that finding a hit to stake is not difficult and setting max_target to 2^{208}).

Next the wallet and the transaction pool are loaded using load_wallet and load_txpool from the commands module (see Chapter 14).

A random 64-bit nonce is generated in order to prevent the node from connecting to itself. The variable this_nodes_nonce from the net module is set to the nonce.

16.2 Initialization of the Network

The initnetwork function starts several threads to handle connections to the network. First, if the configuration variable ip has been set to an IP-address, then a listener socket is opened and a thread is started to listen for incoming connections (using the function netlistener). Then the function netseeker is called which loads the known peers and calls netseeker_loop. The loop tries to connect to peers every 10 minutes, unless the maximum number of connections has been reached.

Each time a connection is created, threads are created for listening to the peer and for sending messages to the peer. Most of this networking code is in the net module which is described in Chapter 7.

16.3 Main Loop

This main loop prints a prompt (see the configuration variable prompt), reads a line and sends this line to do_command. If the user presses CTRL-D, the End_of_file exception is raised resulting in the process exiting. The process will also exit if the user issues the command exit. A command may also raise the exception Exit which will be silently ignored and the loop continues, issuing prompt and waiting for the next command. All other exceptions are displayed to the user before the loop continues.

The supported commands are not fully implemented. The code for executing most commands is in the commands module (see Chapter 14). Here are a few that are currently supported:

- exit - exit qeditas. (CTRL-D also exits.)
- printassets *[ledger root in hex]* - print the assets held at the addresses in the wallet in the ledger with the given root. This only prints the visible assets. Some assets will be missing if the local approximation of the compact tree does not include the assets explicitly. If the ledger root is omitted, then the ledger root of the most recent best block is used.
- importprivkey - import a private key in Qeditas WIF format.
- importbtcprivkey - import a private key in Bitcoin WIF format.
- importwatchaddr - import a Qeditas address for watching assets, without importing the corresponding private key.
- importwatchbtcaddr - import a Bitcoin address as a Qeditas address for watching assets, without importing the corresponding private key.
- importendorsement - import a Bitcoin signed message to allow a Qeditas private key (held in the wallet) to sign for a corresponding Bitcoin address. This is the safe way to claim assets in the initial distribution.
- btctoqedaddr - give the Qeditas address that corresponds to a given Bitcoin address.
- addnode *ip:port [add—remove—onetry]* - explicitly add or remove a connection to a node.
- clearbanned - call clearbanned in net to clear all banned nodes from memory.
- listbanned - list all banned nodes in bannedpeers from net.
- bannode *ip:port* - call bannode in net to add the given node to the set of banned nodes.
- getpeerinfo - list all current connections.
- nettime - print the current network time and median skew.

- printtx *txid* - call printtx in commands to print the tx with the given id.
- printasset *assetid* - call printasset in commands to print the asset with the given id.
- printhconselt *hconseltroot* - call printhconselt in commands to print the hcons element with the given root.
- printctreeelt *ctreeeltroot* - call printctreeelt in commands to print the ctree element with the given root.
- printctreeinfo *ctreehashroot* - call printctreeinfo in commands to print information about the ctree with the given root.
- createsplitlocktx *address assetid numouts lockheight fee [newaddress [newobligationaddress [ledgerroot]]]* - create a transaction with the given asset at the given address as the only input, with the number of outputs indicated where each output has the given lockheight. The values in the outputs are evenly divided (except possibly the last output) after the fee has been subtracted. If no *newaddress* is given, then the output is to the same address and the same address is used in the new obligations. If *newaddress* is given and *newobligationaddress* is not, then the outputs are to *newaddress* and *newaddress* is used in the obligations. If *newaddress* and *newobligationaddress* are given, then the outputs are to *newaddress* and *newobligationaddress* is used in the obligations. If *ledgerroot* is given, then the corresponding ctree is used to look up the asset for the input. (By default, the ledger root of the current best node is used.) Most of the work is done by the function createsplitlocktx in commands.[1]
- signtx *txinhex* - call signtx in commands to (partially or completely) sign the transaction, giving the signed transaction in hex as output.
- savetxtopool *txinhex* - call savetxtopool in commands to save the transaction to the local pool.
- sendtx *txinhex* - call sendtx in commands to send the given transaction to peers.
- bestblock - print information about the current best node (bestnode from blocktree).
- difficulty - print the difficulty target of the current best node (bestnode from blocktree).
- blockchain - print the header hash and ledger root of the last 1000 blocks of the current block chain.

[1] The purpose of this specialized command is to easily split one large asset into several smaller assets which can stake independently. Ultimately, there would need to be a more general command for creating transactions.

- dumpstate *filename* - dump information about the current state to the given file, as a text file. This is for debugging purposes.

Index

Bibliography

[1] Anonymous. The QED Manifesto. In Alan Bundy, editor, *CADE*, volume 814 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 1994.

[2] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Transactions A of the Royal Society*, 363:2351–2375, 2005.

[3] Chad E. Brown. The Egal Manual, September 2014.

[4] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[5] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log*, 5(2):56–68, 1940.

[6] Mike Croteau and Emir Litranab. Proof of stake: Definite. an implementation of constant staking rewards to promote increased network activity, August 2014.

[7] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[8] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In Dieter Gollmann, editor, *FSE*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.

[9] G. Gentzen. Untersuchungen über das logischen Schliessen. *Mathematische Zeitschrift*, 39:405–431, 1936. Translation in: *Collected papers of Gerhard Gentzen*, ed. M. E. Szabo, North-Holland, Amsterdam [1969], pp. 68–131.

[10] go1111111. Any coin that replaces Bitcoin will use the Bitcoin blockchain, 2013.

[11] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[12] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.

[13] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, 1979.

[14] Michael J. C. Gordon. Introduction to the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*, pages 2–3. IEEE Computer Society, 1991.

[15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture, 2015.

[16] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[17] W.A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, New York, 1980. Academic Press.

[18] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

[19] Ralph C. Merkle. Protocols for public key cryptosystems. In *Proc. 1980 Symposium on Security and Privacy*, pages 122–133. IEEE Computer Society, April 1980.

[20] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 475–490. IEEE Computer Society, 2014.

[21] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[23] NIST. FIPS 180-4: Secure Hash Standard (SHS), 2012.

[24] Ulf Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[25] D. Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965.

[26] Peter R. Spin-offs: bootstrap your alt-coin with a bitcoin-blockchain-based initial coin distribution, 2014.

[27] Certicom Research. Sec 2: Recommended elliptic curve domain parameters, 2000. Version 1.0.

[28] Piotr Rudnicki and Andrzej Trybulec. Mathematical knowledge management in mizar. In *Proc. of MKM 2001*, 2001.

[29] Kazuhiko Sakaguchi. `proofmarket.org`.

[30] Bill White. Formal Idealizations of Cryptographic Hashing, 2015.

[31] Bill White. Qeditas: A Formal Library as a Bitcoin Spin-Off, 2015.

[32] Bill White. A Theory for Lightweight Cryptocurrency Ledgers, 2015.