Contents

1 Literate rust programming

1.1 Translating a program from rust to hacspec

We want to translate part of the following piece of code, such that we can use hacspec to reason about it.

```
use concordium_std::{collections::BTreeMap, *};
use core::fmt::Debug;

#[derive(Debug, Serialize, SchemaType, Eq, PartialEq, PartialOrd)]
pub enum AuctionState {
    NotSoldYet,
    Sold(AccountAddress),
}

#[contract_state(contract = "auction")]
#[derive(Debug, Serialize, SchemaType, Eq, PartialEq)]
pub struct State {
    auction_state: AuctionState,
    highest_bid: Amount,
    item: Vec<u8>,
    expiry: Timestamp,
    #[concordium(size_length = 2)]
    bids: BTreeMap<AccountAddress, Amount>,
}

fn fresh_state(itm: Vec<u8>, exp: Timestamp) -> State {
    State {
auction_state: AuctionState::NotSoldYet,
highest_bid: Amount::zero(),
item: itm,
expiry: exp,
bids: BTreeMap::new(),
    }
}

#[derive(Serialize, SchemaType)]
struct InitParameter {
    item: Vec<u8>,
    expiry: Timestamp,
}

#[derive(Debug, PartialEq, Eq, Clone, Reject)]
enum BidError {
    ContractSender,
    BidTooLow,
    BidsOverWaitingForAuctionFinalization,
    AuctionFinalized,
}

#[derive(Debug, PartialEq, Eq, Clone, Reject)]
enum FinalizeError {
    BidMapError,
    AuctionStillActive,
    AuctionFinalized,
}

#[init(contract = "auction", parameter = "InitParameter")]
fn auction_init(ctx: &impl HasInitContext) -> InitResult<State> {
    let parameter: InitParameter = ctx.parameter_cursor().get()?;
    Ok(fresh_state(parameter.item, parameter.expiry))
}

#[receive(contract = "auction", name = "bid", payable)]
fn auction_bid<A: HasActions>(
    ctx: &impl HasReceiveContext,
    amount: Amount,
    state: &mut State,
) -> Result<A, BidError> {
    ensure!(
state.auction_state == AuctionState::NotSoldYet,
BidError::AuctionFinalized
    );
    let slot_time = ctx.metadata().slot_time();
    ensure!(
slot_time <= state.expiry,
BidError::BidsOverWaitingForAuctionFinalization
    );

    let sender_address = match ctx.sender() {
Address::Contract(_) => bail!(BidError::ContractSender),
Address::Account(account_address) => account_address,
    };
    let bid_to_update = state
.bids
.entry(sender_address)
.or_insert_with(Amount::zero);

    *bid_to_update += amount;

    ensure!(*bid_to_update > state.highest_bid, BidError::BidTooLow);
    state.highest_bid = *bid_to_update;

    Ok(A::accept())
}

#[receive(contract = "auction", name = "finalize")]
fn auction_finalize<A: HasActions>(
    ctx: &impl HasReceiveContext,
    state: &mut State,
) -> Result<A, FinalizeError> {
    ensure!(
state.auction_state == AuctionState::NotSoldYet,
FinalizeError::AuctionFinalized
    );

    let slot_time = ctx.metadata().slot_time();
    ensure!(slot_time > state.expiry, FinalizeError::AuctionStillActive);

    let owner = ctx.owner();

    let balance = ctx.self_balance();
    if balance == Amount::zero() {
Ok(A::accept())
    } else {
let mut return_action = A::simple_transfer(&owner, state.highest_bid);
let mut remaining_bid = None;

for (addr, &amnt) in state.bids.iter() {
    if amnt < state.highest_bid {
return_action = return_action.and_then(A::simple_transfer(addr, amnt));
    } else {
ensure!(remaining_bid.is_none(), FinalizeError::BidMapError);
state.auction_state = AuctionState::Sold(*addr);
remaining_bid = Some((addr, amnt));
    }
}

match remaining_bid {
    Some((_, amount)) => {
ensure!(amount == state.highest_bid, FinalizeError::BidMapError);
Ok(return_action)
    }
    None => bail!(FinalizeError::BidMapError),
}
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::sync::atomic::{AtomicU8, Ordering};
    use test_infrastructure::*;
    static ADDRESS_COUNTER: AtomicU8 = AtomicU8::new(0);
    const AUCTION_END: u64 = 1;
    const ITEM: &str = "Starry night by Van Gogh";

    fn dummy_fresh_state() -> State {
dummy_active_state(Amount::zero(), BTreeMap::new())
    }

    fn dummy_active_state(highest: Amount, bids: BTreeMap<AccountAddress, Amount>) -> S
```

```
State {
    auction_state: AuctionState::NotSoldYet,
    highest_bid: highest,
    item: ITEM.as_bytes().to_vec(),
    expiry: Timestamp::from_timestamp_millis(AUCTION_END),
    bids,
}
    }

    fn expect_error<E, T>(expr: Result<T, E>, err: E, msg: &str)
    where
E: Eq + Debug,
T: Debug,
    {
let actual = expr.expect_err(msg);
assert_eq!(actual, err);
    }

    fn item_expiry_parameter() -> InitParameter {
InitParameter {
    item: ITEM.as_bytes().to_vec(),
    expiry: Timestamp::from_timestamp_millis(AUCTION_END),
}
    }

    fn create_parameter_bytes(parameter: &InitParameter) -> Vec<u8> {
to_bytes(parameter)
    }

    fn parametrized_init_ctx<’a>(parameter_bytes: &’a Vec<u8>) -> InitContextTest<’a>
let mut ctx = InitContextTest::empty();
ctx.set_parameter(parameter_bytes);
ctx
    }

    fn new_account() -> AccountAddress {
let account = AccountAddress([ADDRESS_COUNTER.load(Ordering::SeqCst); 32]);
ADDRESS_COUNTER.fetch_add(1, Ordering::SeqCst);
account
    }
    fn new_account_ctx<’a>() -> (AccountAddress, ReceiveContextTest<’a>) {
let account = new_account();
let ctx = new_ctx(account, account, AUCTION_END);
(account, ctx)
    }

    fn new_ctx<’a>(
owner: AccountAddress,
sender: AccountAddress,
slot_time: u64,
    ) -> ReceiveContextTest<’a> {
let mut ctx = ReceiveContextTest::empty();
ctx.set_sender(Address::Account(sender));
ctx.set_owner(owner);
ctx.set_metadata_slot_time(Timestamp::from_timestamp_millis(slot_time));
ctx
    }

    #[test]
    fn test_init() {
let parameter_bytes = create_parameter_bytes(&item_expiry_parameter());
let ctx = parametrized_init_ctx(&parameter_bytes);

let state_result = auction_init(&ctx);
let state = state_result.expect("Contract initialization results in error");
assert_eq!(
    state,
    dummy_fresh_state(),
    "Auction state should be new after initialization"
);
    }

    #[test]
    fn test_auction_bid_and_finalize() {
let parameter_bytes = create_parameter_bytes(&item_expiry_parameter());
let ctx0 = parametrized_init_ctx(&parameter_bytes);

let amount = Amount::from_micro_gtu(100);
let winning_amount = Amount::from_micro_gtu(300);
let big_amount = Amount::from_micro_gtu(500);

let mut bid_map = BTreeMap::new();

let mut state = auction_init(&ctx0).expect("Initialization should pass");

let (alice, alice_ctx) = new_account_ctx();
verify_bid(&mut state, alice, &alice_ctx, amount, &mut bid_map, amount);

verify_bid(
    &mut state,
    alice,
    &alice_ctx,
    amount,
    &mut bid_map,
    amount + amount,
);

let (bob, bob_ctx) = new_account_ctx();
verify_bid(
    &mut state,
    bob,
    &bob_ctx,
    winning_amount,
    &mut bid_map,
    winning_amount,
);

let mut ctx4 = ReceiveContextTest::empty();
ctx4.set_metadata_slot_time(Timestamp::from_timestamp_millis(AUCTION_END));
let finres: Result<ActionsTree, _> = auction_finalize(&ctx4, &mut state);
expect_error(
    finres,
    FinalizeError::AuctionStillActive,
    "Finalizing auction should fail when it’s before auction-end time",
);

let carol = new_account();
let dave = new_account();
let mut ctx5 = new_ctx(carol, dave, AUCTION_END + 1);
```

```
ctx5.set_self_balance(winning_amount);
let finres2: Result<ActionsTree, _> = auction_finalize(&ctx5, &mut state);
let actions = finres2.expect("Finalizing auction should work");
assert_eq!(
    actions,
    ActionsTree::simple_transfer(&carol, winning_amount)
.and_then(ActionsTree::simple_transfer(&alice, amount + amount))
);
assert_eq!(
    state,
    State {
auction_state: AuctionState::Sold(bob),
highest_bid: winning_amount,
item: ITEM.as_bytes().to_vec(),
expiry: Timestamp::from_timestamp_millis(AUCTION_END),
bids: bid_map,
    }
);

let finres3: Result<ActionsTree, _> = auction_finalize(&ctx5, &mut state);
expect_error(
    finres3,
    FinalizeError::AuctionFinalized,
    "Finalizing auction a second time should fail",
);

let res4: Result<ActionsTree, _> = auction_bid(&bob_ctx, big_amount, &mut state);
expect_error(
    res4,
    BidError::AuctionFinalized,
    "Bidding should fail because the auction is finalized",
);
    }

    fn verify_bid(
mut state: &mut State,
account: AccountAddress,
ctx: &ContextTest<ReceiveOnlyDataTest>,
amount: Amount,
bid_map: &mut BTreeMap<AccountAddress, Amount>,
highest_bid: Amount,
    ) {
let res: Result<ActionsTree, _> = auction_bid(ctx, amount, &mut state);
res.expect("Bidding should pass");
bid_map.insert(account, highest_bid);
assert_eq!(*state, dummy_active_state(highest_bid, bid_map.clone()));
    }

    #[test]
    fn test_auction_bid_repeated_bid() {
let (account1, ctx1) = new_account_ctx();
let ctx2 = new_account_ctx().1;

let parameter_bytes = create_parameter_bytes(&item_expiry_parameter());
let ctx0 = parametrized_init_ctx(&parameter_bytes);

let amount = Amount::from_micro_gtu(100);

let mut bid_map = BTreeMap::new();

let mut state = auction_init(&ctx0).expect("Init results in error");

verify_bid(&mut state, account1, &ctx1, amount, &mut bid_map, amount);

let res2: Result<ActionsTree, _> = auction_bid(&ctx2, amount, &mut state);
expect_error(
    res2,
    BidError::BidTooLow, /* { bid: amount, highest_bid: amount } */
    "Bidding 2 should fail because bid amount must be higher than highest bid",
);
    }

    #[test]
    fn test_auction_bid_zero() {
let ctx1 = new_account_ctx().1;
let parameter_bytes = create_parameter_bytes(&item_expiry_parameter());
let ctx = parametrized_init_ctx(&parameter_bytes);

let mut state = auction_init(&ctx).expect("Init results in error");
let res: Result<ActionsTree, _> = auction_bid(&ctx1, Amount::zero(), &mut state);
expect_error(
    res,
    BidError::BidTooLow, /* { bid: Amount::zero(), highest_bid: Amount::zero()} */
    "Bidding zero should fail",
);
    }
}
```

We will end up with two files, one being the rust wrapper and one being the hacspec code.

We have all the usal imports for the file

```
use concordium_std::{collections::BTreeMap, *};
use core::fmt::Debug;
use crate::provider::Action;
```

And then the hacspec speciffic imports

```
use auction::*;
use hacspec_lib::*;
```

and in the hacspec file we only import the hacspec library

```
use hacspec_lib::*;
```

We then translate enums, by first translating all types used by the enum, and then defining the corresponding enum, using the translated types.

```
#[derive(Debug, Serialize, SchemaType, Eq, PartialEq, PartialOrd, Clone)]
pub enum AuctionState {
    NotSoldYet,
    Sold(AccountAddress),
}
```

here we had to change the argument for one of the enums since AccountAddress is not in hacspec. We represent AccountAddress by UserAddress defined as

```
array!(UserAddress, 32, u8);
```

for which we have the following coercion functions

```
fn user_address_to_accout_address(acc: UserAddress) -> AccountAddress {
    AccountAddress([
acc[0], acc[1], acc[2], acc[3], acc[4], acc[5], acc[6], acc[7], acc[8], acc[9], acc[10]
acc[11], acc[12], acc[13], acc[14], acc[15], acc[16], acc[17], acc[18], acc[19], acc[20
acc[21], acc[22], acc[23], acc[24], acc[25], acc[26], acc[27], acc[28], acc[29], acc[30
acc[31],
    ])
}

fn u8x32_to_user_address(acc: [u8; 32]) -> UserAddress {
    UserAddress([
acc[0], acc[1], acc[2], acc[3], acc[4], acc[5], acc[6], acc[7], acc[8], acc[9], acc[10]
acc[11], acc[12], acc[13], acc[14], acc[15], acc[16], acc[17], acc[18], acc[19], acc[20
acc[21], acc[22], acc[23], acc[24], acc[25], acc[26], acc[27], acc[28], acc[29], acc[30
acc[31],
    ])
}
```

we then get the hacspec enum

```
pub enum AuctionState {
    NotSoldYet,
    Sold(UserAddress),
}
```

the enum types also get a pair of coercion functions

```
fn my_auction_state_to_their_auction_state(s: auction::AuctionState) -> AuctionState {
    match s {
auction::AuctionState::NotSoldYet => AuctionState::NotSoldYet,
auction::AuctionState::Sold(a) => AuctionState::Sold(user_address_to_accout_address(a))
    }
}

fn their_auction_state_to_my_auction_state(s: AuctionState) -> auction::AuctionState {
    match s {
AuctionState::NotSoldYet => auction::AuctionState::NotSoldYet,
AuctionState::Sold(a) => auction::AuctionState::Sold(u8x32_to_user_address(a.0)),
    }
}
```

when translating structs we need to convert them to tuples, but again we must first convert the inner types to hacspec types. We convert Amount and Timestamp to u64, while BTreeMaps are converted to a pair of binary sequences, representing a list of key value pairs:

```
pub type SeqMap = (PublicByteSeq, PublicByteSeq);
```

with the coercion functions

```
fn seq_map_to_btree_map(m: SeqMap) -> BTreeMap<AccountAddress, concordium_std::Amount>
    let (m1, m2) = m;

    let m1prime = (0..m1.len() / 32).map(|x| UserAddress::from_seq(&m1.clone().slice(x
    let m2prime =
(0..m2.len() / 8).map(|x| u64_from_be_bytes(u64Word::from_seq(&m2.slice(x * 8, 8))));

    (m1prime.zip(m2prime)).fold(BTreeMap::new(), |mut t, (x, y)| {
t.insert(
    user_address_to_accout_address(x),
    concordium_std::Amount { micro_gtu: y },
);
t
    })
}

fn btree_map_to_seq_map(m: BTreeMap<AccountAddress, concordium_std::Amount>) -> SeqMap
    (
m.keys()
    .map(|x| u8x32_to_user_address(x.0))
    .fold(PublicByteSeq::new(0_usize), |v, x| v.concat(&x)),
m.values()
    .map(|x| x.micro_gtu)
    .fold(PublicSeq::new(0_usize), |v, x| {
v.concat(&u64_to_be_bytes(x))
    }),
    )
}
```

and we have to implement the functions for the data structure in hacspec

```
pub enum MapEntry {
    Entry(u64, SeqMap),
}
```

```
fn seq_map_entry(m: SeqMap, sender_address: UserAddress) -> MapEntry {
    let (m1, m2) = m.clone();
    let mut res = MapEntry::Entry(
0_u64,
(
    m1.clone().concat(&sender_address),
    m2.clone().concat(&u64_to_be_bytes(0_u64)),
),
    );

    for x in 0..m1.clone().len() / 32 {
if UserAddress::from_seq(&m1.clone().slice(x * 32, 32)) == sender_address {
    res = MapEntry::Entry(
u64_from_be_bytes(u64Word::from_seq(&m2.slice(x * 8, 8))),
m.clone(),
    );
}
    }

    res
}

pub enum MapUpdate {
    Update(u64, SeqMap),
}

fn seq_map_update_entry(m: SeqMap, sender_address: UserAddress, amount: u64) -> MapUpda
    let (m1, m2) = m;

    let mut res = MapUpdate::Update(
amount,
(
    m1.concat(&sender_address),
    m2.concat(&u64_to_be_bytes(amount)),
),
    );

    for x in 0..m1.clone().len() / 32 {
if UserAddress::from_seq(&m1.clone().slice(x * 32, 32)) == sender_address {
    res = MapUpdate::Update(
amount,
(
    m1.clone().update(x * 32, &sender_address),
    m2.clone().update(x * 8, &u64_to_be_bytes(amount)),
),
    );
}
    }

    res
}
```

Then the struct

```
#[contract_state(contract = "auction")]
#[derive(Debug, Serialize, SchemaType, Eq, PartialEq)]
pub struct State {
    auction_state: AuctionState,
    highest_bid:   concordium_std::Amount,
    item:          Vec<u8>,
    expiry:        concordium_std::Timestamp,
    #[concordium(size_length = 2)]
    bids:          BTreeMap<AccountAddress, concordium_std::Amount>,
}
```

simply becomes the type

```
pub type State = (AuctionState, u64, Seq<u8>, u64, SeqMap);
```

we again define a pair of coercion functions

```
fn my_state_to_their_state(s: auction::State) -> State {
    let (a, b, c, d, e) = s;
    State {
auction_state: my_auction_state_to_their_auction_state(a),
highest_bid: concordium_std::Amount { micro_gtu: b },
item: c.native_slice().to_vec(),
expiry: concordium_std::Timestamp::from_timestamp_millis(d),
bids: seq_map_to_btree_map(e),
    }
}
```

```
fn their_state_to_my_state(s: &mut State) -> auction::State {
    (
their_auction_state_to_my_auction_state(s.auction_state.clone()),
s.highest_bid.micro_gtu,
Seq::from_vec(s.item.clone()),
s.expiry.timestamp_millis(),
btree_map_to_seq_map(s.bids.clone()),
    )
}
```

Then for each function we translate it fully to hacspec, and do coercion

```
fn fresh_state(itm: Vec<u8>, exp: concordium_std::Timestamp) -> State {
    my_state_to_their_state(auction::fresh_state(
Seq::from_vec(itm),
exp.timestamp_millis(),
    ))
}
```

the translated function is

```
pub fn fresh_state(itm: Seq<u8>, exp: u64) -> State {
    (
AuctionState::NotSoldYet,
0_u64,
itm,
exp,
(PublicByteSeq::new(0_usize), PublicByteSeq::new(0_usize)),
    )
}
```

Next we have another struct which is not translated since it is only used to define the input structure of auction$_{\text{init}}$

```
#[derive(Serialize, SchemaType)]
pub struct InitParameter {
    item: Vec<u8>,
    expiry: concordium_std::Timestamp,
}

#[init(contract = "auction", parameter = "InitParameter")]
pub fn auction_init(ctx: &impl HasInitContext) -> InitResult<State> {
    let parameter: InitParameter = ctx.parameter_cursor().get()?;
    Ok(fresh_state(parameter.item, parameter.expiry))
}
```

Here a context is used and passed around so we need to represent this somehow in hacspec, which we do by making a type for each set of relevant context variables we want to take as input or return

```
pub type Context = (u64, UserAddressSet);
```

again we need to define coercions for this (however we only need one direction, as the context is never updated)

```
fn their_context_to_my_context(ctx: &impl HasReceiveContext) -> auction::Context {
    (
ctx.metadata().slot_time().timestamp_millis(),
match ctx.sender() {
    Address::Contract(_) => UserAddressSet::UserAddressNone,
    Address::Account(account_address) => {
UserAddressSet::UserAddressSome(u8x32_to_user_address(account_address.0), ())
    }
},
    )
}
```

We then define some return / error types

```
#[derive(Debug, PartialEq, Eq, Clone, Reject)]
pub enum BidError {
    ContractSender,
    BidTooLow,
    BidsOverWaitingForAuctionFinalization,
    AuctionFinalized,
}
```

which translate direcly to hacspec

```
pub enum BidError {
    ContractSender,
    BidTooLow,
    BidsOverWaitingForAuctionFinalization,
    AuctionIsFinalized,
}
```

Since this is only used once we do the coercion inline in the auction$_{\text{bid}}$ function

```
#[receive(contract = "auction", name = "bid", payable)]
pub fn auction_bid<A: HasActions>(
    ctx: &impl HasReceiveContext,
    amount: concordium_std::Amount,
    state: &mut State,
) -> Result<A, BidError> {
    let (new_state, res) = auction::auction_bid(
their_context_to_my_context(ctx),
amount.micro_gtu,
their_state_to_my_state(state),
    );
    *state = my_state_to_their_state(new_state);

    match res {
Ok(_) => Ok(A::accept()),
Err(auction::BidError::ContractSender) => Err(BidError::ContractSender),
Err(auction::BidError::BidTooLow) => Err(BidError::BidTooLow),
Err(auction::BidError::BidsOverWaitingForAuctionFinalization) => {
    Err(BidError::BidsOverWaitingForAuctionFinalization)
}
Err(auction::BidError::AuctionIsFinalized) => Err(BidError::AuctionFinalized),
    }
}
```

and the implementation in hacspec is

```
pub enum Boolean {
    True,
    False,
}

pub fn auction_bid(ctx: Context, amount: u64, state: State) -> (State, AuctionBidResult
    // ensure!(state.auction_state == AuctionState::NotSoldYet, BidError::AuctionFinali
    let (auction_state, b, c, expiry, e) = state;
    let (slot_time, sender) = ctx;

    let ((acs, upb, ce, expirye, (updated1_mape, updated2_mape)), rese) = match auction
AuctionState::NotSoldYet => match if slot_time <= expiry {
    Boolean::True
} else {
    Boolean::False
} {
    Boolean::True => match sender {
UserAddressSet::UserAddressNone => (
    (auction_state, b, c, expiry, e),
    AuctionBidResult::Err(BidError::ContractSender),
),
UserAddressSet::UserAddressSome(sender_address, _) => {
    match seq_map_entry(e.clone(), sender_address) {
MapEntry::Entry(bid_to_update, new_map) => match seq_map_update_entry(
    new_map.clone(),
    sender_address,
    bid_to_update + amount,
) {
    MapUpdate::Update(updated_bid, updated_map) => match if updated_bid > b
    {
Boolean::True
    } else {
Boolean::False
    } {
Boolean::True => (
    (auction_state, updated_bid, c, expiry, updated_map),
    AuctionBidResult::Ok(()),
),
Boolean::False => (
    (auction_state, b, c, expiry, updated_map),
    AuctionBidResult::Err(BidError::BidTooLow),
),
    },
},
    }
}
    },
    Boolean::False => (
(auction_state, b, c, expiry, e),
AuctionBidResult::Err(BidError::BidsOverWaitingForAuctionFinalization),
    ),
},
AuctionState::Sold(_) => (
    (auction_state, b, c, expiry, e),
    AuctionBidResult::Err(BidError::AuctionIsFinalized),
),
    };

    (
(acs, upb, ce, expirye, (updated1_mape, updated2_mape)),
rese,
    )
}
```

we continue with the context for the finalize function

```
pub type FinalizeContext = (u64, UserAddress, u64);
```

with translation function

```
fn their_context_to_my_finalize_context(ctx: &impl HasReceiveContext) -> auction::Final
    (
ctx.metadata().slot_time().timestamp_millis(),
u8x32_to_user_address(ctx.owner().0),
ctx.self_balance().micro_gtu,
    )
}
```

The error types for the finalize function

```
#[derive(Debug, PartialEq, Eq, Clone, Reject)]
pub enum FinalizeError {
    BidMapError,
    AuctionStillActive,
    AuctionFinalized,
}
```

translates to

```
pub enum FinalizeError {
    BidMapError,
    AuctionStillActive,
    AuctionFinalized,
}
```

which is again handled inline. We then define the some intermediate tyeps and the result type

```
pub enum BidRemain {
    None,
    Some(u64, ()),
}

pub enum FinalizeAction {
    Accept,
    SimpleTransfer(UserAddress, u64, PublicByteSeq),
}

pub type AuctionFinalizeResult = Result<FinalizeAction, FinalizeError>;
```

then the finalize wrapper function becomes

```
#[receive(contract = "auction", name = "finalize")]
pub fn auction_finalize<A: HasActions>(
    ctx: &impl HasReceiveContext,
    state: &mut State,
) -> Result<A, FinalizeError> {
    let (new_state, res) = auction::auction_finalize(
their_context_to_my_finalize_context(ctx),
their_state_to_my_state(state),
    );
    *state = my_state_to_their_state(new_state);

    match res {
Ok(FinalizeAction::Accept) => Ok(A::accept()),
Ok(FinalizeAction::SimpleTransfer(owner, b, s)) => Ok((0..s.len() / (32 + 8)).fold(
    A::simple_transfer(
&user_address_to_accout_address(owner),
concordium_std::Amount { micro_gtu: b },
    ),
    |t, x| {
t.and_then(A::simple_transfer(
    &user_address_to_accout_address(UserAddress::from_seq(
&s.slice(x * (32 + 8), 32),
    )),
    concordium_std::Amount {
```

```
micro_gtu: u64_from_be_bytes(u64Word::from_seq(
    &s.slice(x * (32 + 8) + 32, 8),
)),
    },
))
    },
)),
Err(auction::FinalizeError::BidMapError) => Err(FinalizeError::BidMapError),
Err(auction::FinalizeError::AuctionStillActive) => Err(FinalizeError::AuctionStillActiv
Err(auction::FinalizeError::AuctionFinalized) => Err(FinalizeError::AuctionFinalized),
    }
}
```

with the hacspec translation

```
pub fn auction_finalize(ctx: FinalizeContext, state: State) -> (State, AuctionFinalizeR
    let (mut auction_state, b, c, expiry, (m1, m2)) = state;
    let (slot_time, owner, balance) = ctx;

    let (continues, mut return_action) = match auction_state {
AuctionState::NotSoldYet => {
    if slot_time > expiry {
if balance == 0_u64 {
    (false, AuctionFinalizeResult::Ok(FinalizeAction::Accept))
} else {
    (
true,
AuctionFinalizeResult::Ok(FinalizeAction::SimpleTransfer(
    owner,
    b,
    PublicByteSeq::new(0_usize),
)),
    )
}
    } else {
(
    false,
    AuctionFinalizeResult::Err(FinalizeError::AuctionStillActive),
)
    }
}
AuctionState::Sold(_) => (
    false,
    AuctionFinalizeResult::Err(FinalizeError::AuctionFinalized),
),
    };

    let mut remaining_bid = BidRemain::None;

    if continues {
for x in 0..m1.clone().len() / 32 {
    let amnt = u64_from_be_bytes(u64Word::from_seq(&m2.slice(x * 8, 8)));
    let addr = UserAddress::from_seq(&m1.clone().slice(x * 32, 32));
    if amnt < b {
return_action = match return_action {
    AuctionFinalizeResult::Ok(a) => match a {
FinalizeAction::SimpleTransfer(o, b, a) => {
    AuctionFinalizeResult::Ok(FinalizeAction::SimpleTransfer(
o,
b,
a.concat(&addr).concat(&u64_to_be_bytes(amnt)),
    ))
}
FinalizeAction::Accept => AuctionFinalizeResult::Ok(FinalizeAction::Accept),
    },
    AuctionFinalizeResult::Err(e) => AuctionFinalizeResult::Err(e),
};
    } else {
if match remaining_bid {
    BidRemain::None => true,
    BidRemain::Some(_, _) => false,
} {
    auction_state = AuctionState::Sold(addr);
    remaining_bid = BidRemain::Some(amnt, ());
} else {
    return_action = AuctionFinalizeResult::Err(FinalizeError::BidMapError);
}
    }
}
    };
    if continues {
return_action = match remaining_bid {
    BidRemain::Some(amount, _) => match if amount == b {
Boolean::True
    } else {
Boolean::False
    } {
Boolean::True => return_action,
Boolean::False => AuctionFinalizeResult::Err(FinalizeError::BidMapError),
    },
    BidRemain::None => AuctionFinalizeResult::Err(FinalizeError::BidMapError),
};
    };

    ((auction_state, b, c, expiry, (m1, m2)), return_action)
}
```

1.2 TODO: Describe tests