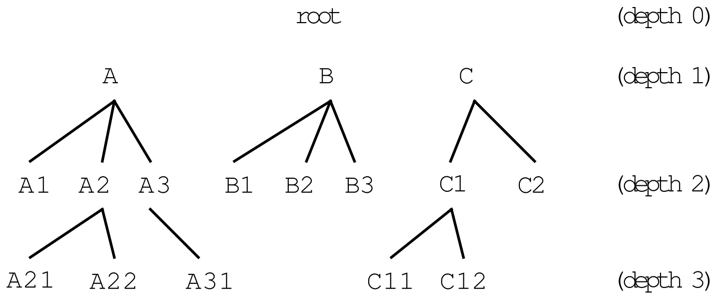
6. Hierarchical Accounts Example

This chapter presents a more complex example of Joule programming, a hierarchical bank account. Hierarchical bank accounts are part of agoric resource management; they implement hierarchical ownership and drawing authority. The account is hierarchical because it can have multiple sub-accounts, each of which is budgeted drawing power on the parent account (and each of which is itself a hierarchical account).

The importance of hierarchical ownership and drawing authority is explained in Section 9.1.

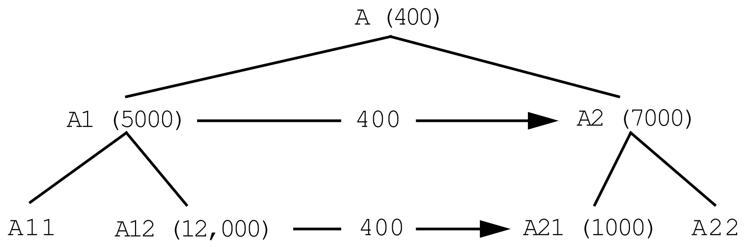
Fig. 6.1 Tree of hierarchical accounts



The root server is not an account but the environment in which top-level accounts are created. Each top-level account can be thought of as the supply of a single currency. In this model, there is no exchange between currencies; each is completely separate.

A hierarchical account can create sub-accounts with arbitrary balances. The balances an account may assign to its subaccounts are unlimited. When a sub-account within that account needs to transfer funds outside of the parent account, however, the amount is limited by the balance of the parent account. This is because the balances of their

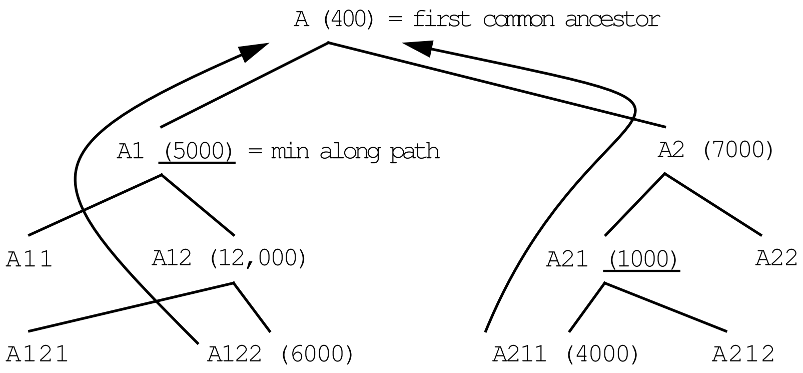
Fig. 6.2 Transfer of funds



respective parent accounts must be balanced as well. In Figure 6.2, any amount (up to the balance of A11) can be transferred from A11 to A12, because these are totally internal to the A1 parent account; however, the transfer of 400 credits from A12 to A21 must be covered by a corresponding transfer from A12's parent A1 to A21's parent A2. The maximum amount for such a move is A1's balance of 5,000 tokens.

In general, the amount that can be transferred from one account to another anywhere in the hierarchy is the minimum of the local balances of the accounts on the path from the donor account to the nearest ancestor it has in common with the recipient account (not including the common ancestor account itself).

Fig. 6.3 Nearest common ancestor for two accounts



For example, in Figure 6.3, the most that could be transferred from A122 to A2 or any of its descendants is 5,000 tokens, the minimum among the local balances of A122 and its ancestors A12 and A1. The most that could be transferred from A211 to A1 or any of its subaccounts is 1,000, the minimum of the balances of A211, A21, and A2.

The term *fractal reserve banking* has been applied to this hierarchical system of accounts. The system is "fractal" because it applies the device of fractional reserve banking recursively. The logical relationship of pieces to wholes does not change at different levels of granularity—the system exhibits the fractal property of *self-similarity*.

6.1. Hierarchical Accounts Components

6.1.1. Type Definitions

To program such a system of Account servers in Joule, we first define the type Account. Any server claiming to be of type Account must accept the set of requests specified by this **Type** form:

```
Type Account
    super Basic
    op split: amount account>
    op deposit: account amount deposited?>
    op budget: amount account>
    op balance: max account balance>
    op private: priv>
endType
```

The split: request will instruct the account to create a sibling account and transfer amount from its own balance to the new account. The result revealed is the public channel to the new account. Because this new account is created by its sibling, its balance must be deducted from the balance of the existing sibling account; money is conserved among siblings. The budget: request instructs the account to create a new subaccount, with an initial balance of amount, which (since it is internal money) can be arbitrary. The deposit: request transfers amount from an existing account to the account receiving the request.

The balance: request takes three arguments: an amount, another account, and a result channel. The balance: request addresses the question "Could this account transfer max tokens into account?" The result revealed is the minimum of max and the maximum amount available for such a transfer (which is the minimum of all the balances of ancestors from the queried account to the ancestor it has in common with account). The candidate amount max is present to avoid infinities in the protocol.

The second **Type** form defines the *private requests* any Account should accept:

```
Type AccountPrivate
    super Basic
    op public: pub>
    op depth: depth>
    op parent: parent>
    op balance: max ancestor balance>
    op reserve: amount ancestor commit? success>
endType
```

Private methods can only be activated by requests received on the server's private channel. A server can receive from any number of channels; *private channels* are closely held because they accept messages with special capabilities. The same message, received via private and public channels to a server, might produce completely different behavior. The private requests to an Account server are used for special functions which should be kept secure.

The public: request reveals the acceptor for a public channel to the server. This ensures that any server which has access to the private channel of an Account can send messages to its public channel as well.

The depth: request reveals how far down in the account hierarchy this account is. It is used only for finding the first common ancestor of two accounts. The parent: request reveals an acceptor for the private channel of this account's parent. The private balance: requests are used to implement the public balance: requests.

The reserve: request instructs the account to adjust its own balance to reflect an impending withdrawal. This adjustment is conditionally based on the commit? flag passed to it. The success> distributor is used to signal success or failure to the server which sent the reserve: request.

The "?" suffix is conventional in Joule to indicate a flag, a port to a Boolean value (true or false).

6.1.2. The make-account Server

The procedure make-account creates new Account servers. Nested within it is the **Server** Account form that defines the behavior of the created accounts.

```
Server make-account :: amount parent account>
    • account> –> account
    Server account
Server account
    var myLocalBalance = amount
    var myParent = parent
    var myDepth = (parent depth:) + 1
    implements Account
```

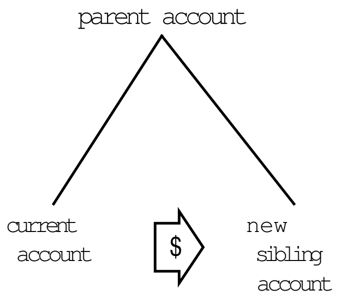
The new server account is created using the parameters passed with the "::" request to make-account. The result distributor account> corresponds to an acceptor held by the server that called make-account; that server can thus send to the new account.

The new account is created with three instance variables. myLocalBalance has the initial value amount provided in the call to make-account. The parent myParent of the new account is specified by the supplied acceptor parent. This acceptor must be for the private channel of the parent account because of the special information subaccounts need about their ancestors (for example, the depth: request, needed to determine common ancestors, is private).

6.1.2.1. The split: Request

The **op** extensions to the **Server** form define the methods of the account. The split: request creates a sibling account:

Fig. 6.4 split: creates new sibling account



```
op split: amount account>
    Define balance
        If amount < 0
            • balance> -> myLocalBalance
            Signal positive-amount-required: amount
        orIf amount > myLocalBalance
            • balance> -> myLocalBalance
            Signal insufficient-funds: myLocalBalance
        else
            • balance> -> myLocalBalance - amount
            • make-account :: amount myParent account>
        endIf
    endDefine
    set myLocalBalance balance
```

The **Define** statement creates a channel balance which can be used immediately by the **set** statement to change (if necessary) the account's local balance. The statements of a Joule program execute concurrently. The instance variable myLocalBalance can be set to balance before the server that will receive from balance is known. If some other computation sends to myLocalBalance before balance is defined, those messages will wait in the channel until the server that should receive them is determined.

Meanwhile, the **If** guards race to evaluate. If the creation of the account fails because a negative initial balance was specified for the new account, or because the current account does not contain enough tokens to provide the requested initial balance for the sibling, then `balance` sends to `myLocalBalance` (meaning that `myLocalBalance` ends up unchanged), and the appropriate exception is **Signal**ed.

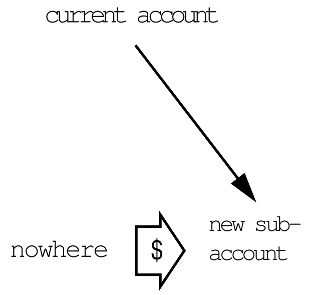
Since accounts happen to never have negative values, the **If** is actually a determinate choice.

Otherwise, the initial balance of the new account is deducted from the present balance of this account, and `make-account` is sent the "::" request to create the new account. Since it is a sibling of this account, it has the same parent (and is passed the private channel to that parent).

6.1.2.2. The `budget:` Request

The method for the `budget:` request is even simpler. Creation of a subaccount has no effect on the local balance of the current account, so we only need to check that the initial balance requested is non-negative. The request to `make-account` is straightforward:

Fig. 6.5 `budget:` creates a new subaccount, with any balance



```
op budget: amount account>
    If amount < 0
        Signal positive-amount-required: amount
    else
        • make-account :: amount Private account>
    endIf
```

Because the new subaccount must have private access to its parent (the current account), the acceptor for this account's `Private` channel is passed in the request to `make-account`.

6.1.2.3. The `balance:` Request

The public `balance:` request "passes the buck" to its private counterpart.

```
reveal the balance of the receiver with respect to the ancestor in common with supplied
        account.
    op balance: max account balance>
        Define
            common =
                common-ancestor :: Private (private :: account)
        endDefine
        to Private balance: max common balance>
```

6.1.2.4. The `private:` Request

The `private`: request instructs the server to reveal its private channel.

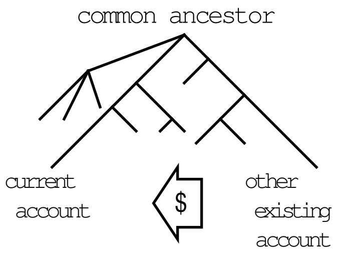
```
op private: priv>
    • priv> -> Private
```

Sending the `private`: request to the public channel forwards the supplied distributor to the private channel. This implementation is clearly insecure. The methods enabling a server to decide securely whether or not to reveal its private channel (using a SealedEnvelope) will be discussed in Section 8.2.1.

6.1.2.5. The `deposit:` and `reserve:` Requests

The `deposit`: request transfers an amount from another account to this account. Whether or not the deposit attempt succeeded is revealed on the result channel `deposited?>`. Before the `deposit`: method can proceed with the transfer, it needs to ensure that the donor account is actually able to transfer that amount. It does this by sending the `reserve`: request to the private channel of the donor account.

Fig. 6.6 `deposit`: transfers money from another existing account



```
op deposit: account amount deposited?>
    Define
        accPriv = private :: account,
        common = common-ancestor :: Private accPriv,
        withdrawn? =
            accPriv reserve: amount transferAmt common
    endDefine
    • deposited?> -> withdrawn?
    Define transferAmt
        If withdrawn? & amount >= 0
            • transferAmt> -> amount
        else
            • transferAmt> -> 0
        endIf
    endDefine
    Define ignore> endDefine
    to Private reserve: 0 (transferAmt negated:) common ignore>
```

The `deposit`: method accepts three arguments: `account`, from which the deposit is being transferred; the `amount` of the deposit, and a result flag `deposited?`, letting the depositor know that the deposit succeeded.

The first **Define** statement calls the private server to get the private channel of the depositing account. Again, this version of private does not implement real Joule security techniques.

```
    Reveal the private channel of account. This procedure will be substituted later for one
        that is secure.
    Server private :: account priv>
        • account private: priv>
    endServer
```

In response to the "::" message, private sends the request private: priv> to account's public channel; account then forwards priv> to account's private channel.

Back in the **op** deposit: block, accPriv is an acceptor for the depositor's private channel. The next **Define** block calls the common-ancestor procedure. In response to the "::" request, common-ancestor forwards the distributor ancestor> to the closest common ancestor of the acc1 and acc2 accounts. It does this by calling itself recursively: if the depth of one account is greater than the other, it recurs with the shallower account and the parent of the deeper account as arguments. If both accounts are of the same depth, it recurs with their two parents. This continues until the new arguments are (acceptors for) the same account.

```
Server common-ancestor :: acc1 acc2 ancestor>
    Define d1 = acc1 depth: , d2 = acc2 depth: endDefine
    If d1 < d2
        • common-ancestor :: acc1 (acc2 parent:) ancestor>
    orIf d1 > d2
        • common-ancestor :: (acc1 parent:) acc2 ancestor>
    orIf (d1 = d2) & (acc1 != acc2)
        • common-ancestor :: (acc1 parent:) (acc2 parent:) ancestor>
    orIf acc1 = acc2
        • ancestor> → acc1
    endIf
endServer
```

The **facet** Private extension to the **Server** form introduces the private methods of the Account server. All **op**s following **facet** Private, are private methods.

The next **Define** statement sets withdrawn? to the success flag of the statement accPriv reserve: amount transferAmt common. This statement sends the private request reserve: to the private channel of the depositing account, asking it to verify that it can in fact transfer the amount requested.

The **set** statement can immediately tell myLocalBalance to deliver to newBalance—that is, either the same value it presently has, or its present value minus transferAmt. Again, messages to myLocalBalance will be held and delivered after the new value of myLocalBalance is determined.

What is the value of transferAmt? It is set in the deposit: method—if the flag withdrawn? indicates that the money was reserved as requested, transferAmt is set to the amount specified in the original deposit: request. If withdrawn? indicates that the depositor was unable to reserve the amount requested, then transferAmt is set to zero, and the depositor's local balance does not change.

This is one of the powerful benefits of Joule's inherent concurrency. The deposit: method of the server receiving the deposit sends the reserve:

```
facet Private
    type AccountPrivate
    op reserve: reserveAmt transferAmt ancestor success?>
        Define newBalance, parent'
            If (reserveAmt <= myLocalBalance) &
                    (Private != ancestor)
                • myParent reserve: reserveAmt transferAmt
                                    ancestor success?> then
                                    parent'>
                • newBalance> → myLocalBalance - transferAmt
            else
                • parent'> → myParent
                • newBalance> → myLocalBalance
                • success?> → Private = ancestor
            endIf
        set myParent parent’
        set myLocalBalance newBalance
```

request to the depositing server with an argument transferAmt that does not yet have a value. The depositing server can determine, based on the other arguments of the request, whether or not the request can succeed, and can inform the receiver of this (via the result channel success?>). Based on this go/no-go result flag, the server which sent the reserve: request can now supply the value of transferAmt. Meanwhile, both servers have already used transferAmt to adjust their own local balances.

If the depositor is an ancestor of the receiver, then the depositor does not adjust its own balance—the transfer is entirely internal to the ancestor and does not affect the ancestor's local balance. The withdrawn? flag is set to true, but no money is subtracted from the ancestor's balance.

Both deposit: and reserve: recur to the respective parent accounts, because those balances must also be adjusted by the amount of the transfer, up to but not including the common ancestor of the two accounts. To that common ancestor, the transfer of monies is completely internal, but to every intermediate account, the transfer is real money.

6.1.2.6. Other Private Requests

The other private methods of Account are fairly straightforward:

```
op public: pub>
    • pub> → account
op depth: depth>
    • depth> → myDepth
op parent: parent>
    • parent> → myParent
```

Any server holding the private channel to this account should presumably be allowed to hold the public channel as well; the private request public: reveals it. The depth: and parent: requests are used only by common-ancestor.

6.1.2.7. The Private `balance:` Request

The private `balance:` request reveals the balance of the receiver with respect to an account known to be its ancestor. (Normally, this will be called with the result revealed by `common-ancestor`.)

```
        op balance: max ancestor balance>
            Define parent'
                If ancestor = Private
                    • balance> → max
                    • parent'> → myParent
                else
                    Define
                        localBal = myLocalBalance min: max
                    endDefine
                    • myParent balance: localBal ancestor balance>
                        then parent'>
                endIf
            endDefine
            set myParent parent'
    endServer
endServer
```

The **If** guard `ancestor = Private` halts the recursive passing of `balance:` requests up the tree when they reach the ancestor itself. The **then** extension to the sending of `balance:` to `myParent` is there to ensure that messages from an account to its parent arrive in the order in which they were sent. (If you deposit a sum of money into an empty account, then try to withdraw some of it, the withdrawal attempt will fail unless the order of the requests is preserved.)

The **set** and **Define** statements are running concurrently. The **set** reassigns `myParent` to the acceptor `parent'` created by **Define**. All messages sent to `myParent` are forwarded into the channel `parent'` and held there. The **then** statement is an extension to the message-send statement. It takes as its argument a distributor whose messages (if any) will be forwarded to the target of the send, guaranteed to arrive *after* the one sent in the original message. In this case, the target is `myParent`, and the distributor is `parent'>`, which is holding the messages meant for `myParent` that piled up behind the privileged message `balance: localBal ancestor balance>`. If the other clause of the **If** wins and the **Define** is never executed, then `parent'>` and all the messages in it are forwarded directly to `myParent` in the ordinary Joule fashion, without any ordering.

6.1.3. The `root` Server

Recursive requests that are passed all the way up the "money tree" bottom out at the server `root`, which is the "parent" of the top-level

```
Server root
    op mint: amount account>
        If amount < 0
            Signal positive-amount-required:
        else
            • make-account :: amount Private account>
        endIf
    facet Private
    type AccountPrivate
```

```
        op public: pub>
            Signal not-a-currency:
        op depth: depth>
            • depth> → 0
        op parent: parent>
            Signal broken-recursion:
        op balance: max ancestor balance>
            Signal different-currencies:
        op reserve: amount ancestor commit? success>
            Signal different-currencies:
            • success> → false
    endServer
```

accounts. Except for the `mint`: request, it accepts only private messages—the same set of private messages as `Account`, so its private facet is also of type `AccountPrivate`. The public `mint`: request creates a new currency (a top-level account), with the money supply `amount`, and reveals that account's public channel on `account>`. (Note that `root` signals an exception to the `reserve`: request—once a currency is created, its total money supply cannot be increased.

Here are uninterrupted program listings for the `make-account`, `common-ancestor`, `private`, and `root` servers:

6.2. Program Listings

6.2.1. make-account

```
Server make-account :: amount parent account>
    • account> → account
    Server account
        var myLocalBalance = amount
        var myParent = parent
        var myDepth = (parent depth:) + 1
        type Account
        op split: amount account>
            Define balance
                If amount < 0
                    • balance> → myLocalBalance
                    Signal positive-amount-required: amount
                orIf amount > myLocalBalance
                    • balance> → myLocalBalance
                    Signal insufficient-funds: myLocalBalance
                else
                    • balance> → myLocalBalance - amount
                    • make-account :: amount myParent account>
                endIf
            endDefine
            set myLocalBalance balance
        op budget: amount account>
            If amount < 0
                Signal positive-amount-required: amount
            else
                • make-account :: amount Private account>
            endIf
        op balance: max account balance>
            Define
                common =
                common-ancestor :: Private (private :: account)
            endDefine
            to Private balance: max common balance>
        op deposit: account amount deposited?>
            Define
                accPriv = private :: account,
                common = common-ancestor :: Private accPriv,
                withdrawn? =
                    accPriv reserve: amount transferAmt common
            endDefine
            • deposited?> → withdrawn?
            Define transferAmt
                If withdrawn? & amount >= 0
                    • transferAmt> → amount
                else
                    • transferAmt> → 0
                endIf
            endDefine
            Define ignore> endDefine
            to Private reserve: 0 (transferAmt negated:) common ignore>
        op private: priv>
            • priv> → Private
    facet Private
        type AccountPrivate
        op reserve: reserveAmt transferAmt ancestor success?>
            Define newBalance, parent'
                If (reserveAmt <= myLocalBalance) &
                        (Private != ancestor)
                    • myParent reserve: reserveAmt transferAmt ancestor
                                        success?> then parent'>
                    • newBalance> → myLocalBalance - transferAmt
                else
                    • parent'> → myParent
                    • newBalance> → myLocalBalance
                    • success?> → Private = ancestor
                endIf
            set myParent parent’
            set myLocalBalance newBalance
        op public: pub>
            • pub> → account
        op depth: depth>
            • depth> → myDepth
        op parent: parent>
            • parent> → myParent
        op balance: max ancestor balance>
            Define parent'
                If ancestor = Private
                    • balance> → max
                    • parent'> → myParent
                else
                    Define
                        localBal = myLocalBalance min: max
                    endDefine
                    • myParent balance: localBal ancestor balance>
                        then parent'>
                endIf
            endDefine
            set myParent parent'
    endServer
```

```
endServer
```

6.2.2. common-ancestor

```
Server common-ancestor :: acc1 acc2 ancestor>
    Define d1 = acc1 depth: , d2 = acc2 depth: endDefine
    If d1 < d2
        • common-ancestor :: acc1 (acc2 parent:) ancestor>
    orIf d1 > d2
        • common-ancestor :: (acc1 parent:) acc2 ancestor>
    orIf (d1 = d2) & (acc1 != acc2)
        • common-ancestor :: (acc1 parent:) (acc2 parent:) ancestor>
    orIf acc1 = acc2
        • ancestor> → acc1
    endIf
endServer
```

6.2.3. private

```
Server private :: account priv>
    • account private: priv>
endServer
```

6.2.4. root

```
Server root
    op mint: amount account>
        If amount < 0
            Signal positive-amount-required:
        else
            • make-account :: amount Private account>
        endIf
    facet Private
    type AccountPrivate
    op public: pub>
        Signal not-a-currency:
    op depth: depth>
        • depth> → 0
    op parent: parent>
        Signal broken-recursion:
    op balance: max ancestor balance>
        Signal different-currencies:
    op reserve: amount ancestor commit? success>
        Signal different-currencies:
        • success> → false
endServer
```