Relational Normalization for Programs

Byron Hawkins • Université Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

Problem: Structural Integrity

Background: a denormalized database schema

Usage Scenario

1. Record transactions during the day
2. At night, verify all accounts
3. Archive all records for the day

AccountActivity

account_id	amount	fee	balance
1577158	200.00	2.00	394.57
1577158	-37.00	.37	357.20
1577158	44.00	.44	401.64

Delete Anomaly
After moving records to an archive table, current account balances are lost.

Insert Anomaly
Impossible to insert an account balance without inserting a transaction.

Update Anomaly
Changing a transaction amount requires updating all future balance entries.

Data structures can have the same problems:

```
struct account_activity_t {
  account_t *account;
  double    amount;
  double    fee;
  double    balance;
};

account_activity_t *transactions;
```

```
double get_balance(int account_id) {
  double balance = 0.0;
  for (t = transactions; t; t++) {
    if (t->account->id == account_id)
      balance = t->balance;
  }
  return balance;
}
```

Slow and inconvenient to find an account balance using denormalized data.

Problem: array sequence determines which account balance is the current one.

Functions can exhibit similar problems:

```
account_activity_t *deposit(account_t *account, double amount) {
  account_activity_t *transaction = ALLOC(account_activity_t);
  transaction->account = account;       // applies to all deposits
  transaction->amount = amount;         // applies to all deposits
  transaction->fee = amount * FEE_RATE; // applies to some deposits
  transaction->balance += (amount - transaction->fee);
  return transaction;
}

account_activity_t *online_deposit(account_t *account, double amount) {
  account_activity_t *transaction = deposit(account, amount);
  transaction->balance -= transaction->fee;
  transaction->fee = 0.0;
}
```

This function implements two domain operations that have differing scopes of applicability.

The transaction fee only applies to in-person transactions. Scope mismatch requires `deposit()` to be duplicated or partially reversed.

Definitions and Goals

A normalized application is a normalized composition of normalized components.

Definition: Normalized Composition
Integration of two or more normalized components into a single higher-order normalized component that maintains all functional requirements between the domain entities of the sub-components.

Definition: Normalized Component
An autonomous subsystem within a software application that maintains one functional requirement with respect to a set of domain entities.

Normalization formalizes atomic design.

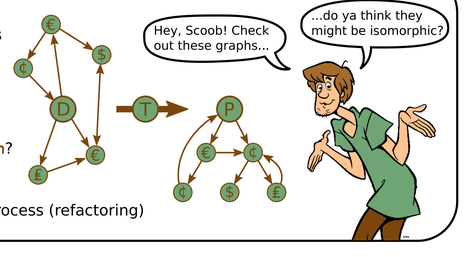
- Goal of normalization: maintain integrity with respect to domain knowledge.
- Challenge of normalization: complexity quickly overwhelms human developers.

Normalization by Graph Transformation

D Domain Graph created by the developer
- Node: Domain Label on source code elements
- Edge: Domain Constraint (e.g., dependency)
- Annotation: Use Case Requirement, e.g.:
  - traverse a set of domain entities via association
  - maximum latency for user interface update

P Program Graph generated by the compiler
- Infer corresponding Use Case Implementation?

T Normalization Graph Transform
- Fails for non-normalized programs
- Counter-examples guide the normalization process (refactoring)



Normalizing Programs

Background: how to normalize a database schema

AccountActivity

id	account_id	amount	fee	balance
217	1577158	200.00	2.00	394.57
218	1577158	-37.00	.37	357.20
219	1577158	44.00	.44	401.64

REFACTOR

Step #1: add a key that uniquely identifies each entry in the table.

Transaction

id	account_id	amount	fee
217	158	200.00	2.00
218	158	-37.00	.37
219	158	44.00	.44

Account

id	balance
155	22.29
157	9211.95
158	401.64

REFACTOR

Step #2: divide the fields into new tables such that each field depends on the key.

Key Insight
Objects that are independent in the domain should be represented by software elements that maintain comparable independence.

First Order Logic
A normalized schema supports composition and modification of domain objects using any simple first order logic (e.g., SQL).

How to normalize data structures for random access

REFACTOR

```
struct transaction_t {
  account_t *account;
  double    amount;
  double    fee;
};
struct account_t {
  double  balance;
};
```

Step #1: divide structs until each field is fully identified by its owner.

REFACTOR

```
struct transaction_t {
  int       id;
  account_t *account;
  double    amount;
  double    fee;
};
struct account_t {
  int     id;
  double  balance;
};
```

Step #2: define an identifier:
- an index field (`id`), or
- an identifier function

REFACTOR

Step #3: define collections that perform well for random access.

```
transaction_t *transactions;

transaction_t *get_transaction(int id) {
    return transactions[id];
}

hashtable_t *accounts;

account_t *get_account(int id) {
    return (account_t *) hash_find(id);
}
```

Identifying domain elements shouldn't be a big mystery!

How to normalize functions for composable concurrency

Performance Bottleneck
1. Transactions received on a network socket
2. Minimum transaction throughput: 1,000/sec.
3. Maximum transaction processing time: 200ms

```
void transaction_server(socket_t *s) {
    transaction_data_t *data;
    while ((data = s->read()) != NULL) {
        transaction_t *t = inflate(data);
        process_transaction(t);
    }
}
```

Problem: conflicting performance profiles having thread-bound data dependencies.

REFACTOR

```
void transaction_server(socket_t *s) {
    transaction_data_t *data;
    while ((data = s->read()) != NULL) {
        pthread_mutex_lock(&queue_lock);
        enqueue_transaction(queue, data);
        pthread_mutex_unlock(&queue_lock);
    }
}
```

```
void transaction_worker() {
    transaction_data_t *data;
    while (data = wait_queue(queue)) {
        transaction_t *t = inflate(data);
        process_transaction(t);
    }
}
```

Resolution: distribute bottleneck consumers across threads under a dedicated lock.

Automated Normalization Assistance

Eclipse Compiler Plugin: Role Normalization and Synthesis

Find RNS on github:

`@InvocationConstraint` Requires every caller to be a member of the specified domains

`@DomainRole.Join` Assigns domains to top-level AST nodes (class, method, enum)

Tool Usage Model
1. Developer annotates the program with labels that represent domain knowledge
2. Compiler plugin raises an error when domain constraints are contradicted

Annotation hierarchy for transactions

```
class GenericTransactions extends DomainRole
class OnlineTransactions extends GenericTransactions
class TellerTransactions extends GenericTransactions
```

Compile error: crossing a domain-knowledge boundary.

```
@InvocationConstraint(domains = TellerTransactions.class)
AccountActivity tellerDeposit(Account account, double amount) {
  AccountActivity transaction = new AccountActivity();
  transaction.account = account;       // applies to all deposits
  transaction.amount = amount;         // applies to all deposits
  transaction.fee = amount * FEE_RATE; // applies to some deposits
  transaction.balance += (amount - transaction.fee);
  return transaction;
}

@DomainRole.Join(membership = OnlineTransactions.class)
AccountActivity *onlineDeposit(Account account, double amount) {
  AccountActivity transaction = tellerDeposit(account, amount);
  transaction.balance -= transaction.fee;
  transaction.fee = 0.0;
}
```

No error after refactoring (supports domain subclasses).

```
@InvocationConstraint(domains = GenericTransactions.class)
AccountActivity deposit(Account account, double amount) {
  AccountActivity transaction = new AccountActivity();
  transaction.account = account;
  transaction.amount = amount;
  transaction.balance += amount;
  return transaction;
}
@InvocationConstraint(domains = TellerTransactions.class)
@DomainRole.Join(membership = TellerTransactions.class)
AccountActivity tellerDeposit(Account account, double amount) {
  AccountActivity transaction = deposit(account, amount);
  transaction.fee = amount * FEE_RATE;
  transaction.balance += (amount - transaction.fee);
  return transaction;
}
@InvocationConstraint(domains = OnlineTransactions.class)
@DomainRole.Join(membership = OnlineTransactions.class)
AccountActivity *onlineDeposit(Account account, double amount) {
  return transaction = deposit(account, amount);
}
```

Benefits of RNS
1. Documents the domain role of each top-level AST node.
2. Maintains domain boundaries after developers forget them.
3. Enforces domain boundaries for developers who don't know.

RNS examples on github

Simple weather browser app

User interface library with a transactional event model.