Fault Tolerance via Idempotence

G. Ramalingam and Kapil Vaswani
Microsoft Research, India
grama,kapilv@microsoft.com

Abstract

Building distributed services and applications is challenging due to the pitfalls of distribution such as process and communication failures. A natural solution to these problems is to detect potential failures, and retry the failed computation and/or resend messages. Ensuring correctness in such an environment requires distributed services and applications to be *idempotent*.

In this paper, we study the inter-related aspects of process failures, duplicate messages, and idempotence. We first introduce a simple core language (based on λ-calculus) inspired by modern distributed computing platforms. This language formalizes the notions of a service, duplicate requests, process failures, data partitioning, and *local* atomic transactions that are restricted to a single store.

We then formalize a desired (generic) correctness criterion for applications written in this language, consisting of *idempotence* (which captures the desired safety properties) and *failure-freedom* (which captures the desired progress properties).

We then propose language support in the form of a monad that automatically ensures failfree idempotence. A key characteristic of our implementation is that it is decentralized and does not require distributed coordination. We show that the language support can be enriched with other useful constructs, such as compensations, while retaining the coordination-free decentralized nature of the implementation.

We have implemented the idempotence monad (and its variants) in F# and C# and used our implementation to build realistic applications on Windows Azure. We find that the monad has low runtime overheads and leads to more declarative applications.

Categories and Subject Descriptions D.4.5 [Operating Systems]: Reliability—Fault-tolerance; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/server, Distributed applications

General Terms Reliability, Languages, Design

Keywords fault tolerance, idempotence, workflow, transaction, monad

1. Introduction

Distributed computing is becoming mainstream. Several modern platforms offer virtualized distributed systems at low entry cost with the promise of scaling out on demand. But distributed computing comes with its own pitfalls, such as process failures, imperfect messaging, asynchrony and concurrency.

Consider the prototypical bank account transfer service in Fig. 1. The goal of the service is to transfer money between bank accounts, potentially in different banks. If the accounts belong to different banks, ensuring that the transfer executes as an *atomic (distributed) transaction* is usually not feasible, and the natural way of expressing this computation is as a *workflow* [10, 20] consisting of two steps, a debit followed by a credit.

What if the process executing the workflow fails in between the debit and credit steps? A natural solution is to detect this failure and ensure that a different process completes the remaining steps of the workflow. A challenging[1] aspect of realizing this solution is figuring out whether the original process failed before or after completing a particular step (either debit or credit). If not done carefully, the debit or credit step may be executed multiple times, leading to further correctness concerns. Services often rely on a central workflow manager to manage process failures during the workflow (using distributed transactions).

Now consider a (seemingly) different problem. Messages sent between the client initiating the transfer and the service may be lost. The only option for a client, when it does not receive a response within some reasonable time, is to resend its request. Yet the client does not want the transfer to occur twice!

In this paper, we study process and communication failures in the context of workflows. The seemingly different problems caused by process and communication failures are, in fact, inter-related. *Idempotence*, a correctness criterion that requires the system to tolerate duplicate requests, is the key to handling both communication and process failures efficiently. Idempotence, when combined with retry, gives us the essence of a workflow, a fault tolerant composition of atomic actions, for free *without the need for distributed coordination*. In the transfer example, a fault tolerant account transfer can be implemented without a central workflow manager if the debit and credit steps can be designed to be idempotent,

Formalizing Failfree Idempotence. In this paper, we introduce a simple core language λ_{FAIL}, inspired by contemporary cloud platforms. This language formalizes process failure, duplicate requests, partitioned data, and *local* transactions. A local transaction provides ACID guarantees but is restricted to access data within a single partition (typically a single server). Computations in λ_{FAIL} are like workflows, but without any fault-tolerance guarantees for the composition (i.e., the computation may fail between transactions).

We then formalize a generic correctness criterion for applications written in λ_{FAIL}. A simple, powerful and tempting criterion is that an application's behavior in the presence of duplicate requests and process failures should be indistinguishable from its behavior in the absence of duplicate requests and failures. We formal-

[1] In general, detecting failures perfectly in an asynchronous, message passing system is impossible [8]. Conservative failure detection can also lead to the same problem of duplicated computation.

POPL'13, January 23–25, 2013, Rome, Italy.

```
let process (request) =
  match request with
  | ("getBalance", (branch, account)) →
      atomic branch {lookup account}
  | ("transfer", (fromBranch, fromAccount, toBranch, toAccount, amt) →
      atomic fromBranch {
        update fromAccount ((lookup fromAccount) − amt)
      };
      atomic toBranch {
        update toAccount ((lookup toAccount) + amt)
      };
      "Transfer complete."
```

Figure 1: A banking service example, in syntactically sugared λ_{FAIL}, that is neither idempotent nor fault-tolerant.

ize a slightly weaker, but more appropriate, correctness criterion, namely *failure-freedom modulo message duplication*. Informally, this criterion permits the system to send duplicate responses. This weakening is appropriate from the perspective of *composition*: if the recipient of the responses can also tolerate duplicate messages, then the sender is freed of the obligation to send the response exactly once.

Automating Idempotence. Next, we address the problem of automatically ensuring idempotence for a service. We present our solution as a monad, the *idempotence* monad. We then show that idempotence, when coupled with a simple retry mechanism, provides a "free" solution to the problem of tolerating process failures, guaranteeing failure-freedom modulo message duplication. We then propose dedicated language support for idempotent computations.

Decentralized Idempotence and Workflow. The idea underlying the idempotence monad is conceptually simple, but tedious to implement manually (i.e., without the monad). Given a unique identifier associated with a computation, the monad essentially adds logging and checking to each effectful step in the workflow to ensure idempotance. An important characteristic of our implementation (of the monad) is that it is designed to work with contemporary distributed storage systems such as key-value tables. Specifically, it does not assume the presence of dedicated storage for logs that can be accessed atomically with each transaction. The monad reuses the underlying store (in this case a key-value table) to simulate a distinct address space for logging.

This leads to a decentralized implementation of idempotence that does not require any centralized storage or any (distributed) coordination between different stores. Thus, the implementation of idempotance preserves the decentralized nature of the underlying computation. This, in turn, leads to a completely decentralized implementation of a (fault-tolerant) workflow, unlike traditional workflow implementations, which use a centralized workflow coordinator and/or a centralized repository for runtime status information.

Extensions. We then extend the idempotence monad with other useful constructs, while preserving the decentralized nature of the construct. One extension allows the application to associate each transaction in a workflow with a compensating action. Another extension allows the application to generate intermediate responses to client requests and then asynchronously retry the requests on client's behalf. This idiom, especially useful in long running computations, frees the client from having to track status of requests, and leads to more responsive clients.

Implementation. We have implemented the idempotence workflow monad in F# targeting the Windows Azure platform. We have implemented several realistic applications using the idempotence workflow monad [2, 4]. We find that the core business logic in these applications can be declaratively expressed using the monad. Our evaluation shows that performance overheads of using the monad over hand-coded implementations are statistically insignificant.

The rest of the paper is organized as follows. In Section 2, we introduce a language λ_{FAIL} and formalize duplicate requests and process failures. We formalize what it means for a λ_{FAIL} application to correctly tolerate duplicate requests and failures. In Section 3, we present the idempotence monad and show how it can be used to tolerate duplicate requests as well as process failures. In Section 4, we describe extensions of the idempotence construct. In Section 5, we evaluate the idempotence monad and our implementation from the perspective of expressiveness, benefits and overheads. Section 6 discusses related work.

2. Failfree Idempotence

In this section we present a language λ_{FAIL} that distils essential elements of distributed computing platforms such as Windows Azure and formalize the concept of failfree idempotence.

2.1 The Language λ_{FAIL}

Informal Overview. A λ_{FAIL} program e represents a service that receives input requests and produces output responses. An input request v is processed by *creating* an agent to evaluate $e\ v$. When the evaluation of $e\ v$ terminates, producing a value v', v' is sent back as the response. Multiple input requests can be processed concurrently, and their evaluation can be interleaved. Shared, mutable, persistent data is stored in *tables*.

Agents. An agent has its own internal state (captured by local variables of the code). An agent may *fail* at any point in time. A failure models problems such as hardware failure, software crashes and reboots . Data stored in tables is *persistent* and is unaffected by agent failures.

Tables. Tables are persistent maps. They provide primitives to *update* the value bound to a key and *lookup* up the value associated with a key. The language provides a *limited* form of an *atomic transaction*, which enables a set of operations on the *same* table to be performed transactionally. Specifically, the construct "`atomic` $t\ e$" evaluates e in the context of table t (lookups and updates are permitted only on table t within e), guaranteeing isolation (no other agent can access t in the middle of e's evaluation) and the all-or-nothing property (no process failure can happen in the middle of e's evaluation).

Example. Fig. 1 presents a simple bank-account-transfer example in syntactically sugared λ_{FAIL}. This example is neither idempotent (e.g., executing the same *transfer* request twice is not equivalent to executing it once) nor fault-tolerant (e.g., if the agent processing a transfer request fails in between the debit and credit steps).

Syntax. Fig. 2 presents the syntax of λ_{FAIL}, which extends λ-calculus with the primitive operations on tables explained above. In the rest of the paper, we will use extensions such as natural numbers, arithmetic operators, and ordered pairs for brevity. These can be encoded in the core language or added to it in the usual fashion. We also use syntactic sugar, such as `lookup` e, where $e \notin \langle \mathsf{Val} \rangle$, as shorthand, e.g., for $(\lambda x.\texttt{lookup}\ x)e$.

Semantic Domains Fig. 2 defines the semantic domains used in the semantics of λ_{FAIL}. Let $\langle \mathsf{Val} \rangle$ denote the set of all basic values: these basically consist of function abstractions. (As usual, natural numbers, string constants, ordered pairs of values, etc. can all be encoded within $\langle \mathsf{Val} \rangle$ or added to it.)

Let $\langle \mathsf{Val} \rangle_{\mathsf{opt}}$ represent the set of *optional* values (of the form `NONE` or `SOME` v). An element of Σ_{ST} (a map from $\langle \mathsf{Val} \rangle$ to

(a) Evaluation Contexts

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

(b) The Set of Evaluation Rules A used to define the standard semantics of λ_{FAIL}.

[INPUT]
$$\frac{v \in \langle \mathsf{Val} \rangle}{\langle e, \mu, \alpha, I, O\rangle \overset{in(v)}{\Rightarrow} \langle e, \mu, \alpha \uplus \{v \triangleright (e\ v)\}, I \cup \{v\}, O\rangle}$$

[OUTPUT]
$$\frac{v_o \in \langle \mathsf{Val} \rangle}{\langle e, \mu, \alpha \uplus \{v_i \triangleright v_o\}, I, O\rangle \overset{out(v_i, v_o)}{\Rightarrow} \langle e, \mu, \alpha, I, O \cup \{(v_i, v_o)\}\rangle}$$

[NORMAL]
$$\frac{\langle \mu, e\rangle \overset{\omega}{\rightsquigarrow} \langle \mu', e'\rangle}{\langle p, \mu, \alpha \uplus \{v \triangleright e\}, I, O\rangle \overset{\epsilon}{\Rightarrow} \langle p, \mu', \alpha \uplus \{v \triangleright e'\}, I, O\rangle}$$

[FAIL]
$$\frac{}{\langle p, \mu, \alpha \uplus \{v \triangleright e\}, I, O\rangle \overset{\epsilon}{\Rightarrow} \langle p, \mu, \alpha, I, O\rangle}$$

[ATOMIC]
$$\frac{\langle \mu[tn], e()\rangle \overset{\omega}{\twoheadrightarrow}^{*} \langle t, v\rangle, v \in \langle \mathsf{Val} \rangle}{\langle \mu, \texttt{atomic}\ tn\ e\rangle \overset{\omega}{\rightsquigarrow} \langle \mu[tn \mapsto t], v\rangle}$$

[UPDATE]
$$\frac{}{\langle t, \texttt{update}\ k\ v\rangle \overset{(t,k,v)}{\twoheadrightarrow} \langle t[k \mapsto (\texttt{SOME}\ v)], 0\rangle}$$

[LOOKUP]
$$\frac{}{\langle t, \texttt{lookup}\ k\rangle \overset{\epsilon}{\twoheadrightarrow} \langle t, t[k]\rangle}$$

[CONTEXT]
$$\frac{\langle t, e\rangle \overset{u}{\twoheadrightarrow} \langle t', e'\rangle}{\langle t, E[e]\rangle \overset{u}{\twoheadrightarrow} \langle t', E[e']\rangle}$$

[CONTEXT]
$$\frac{\langle \mu, e\rangle \overset{\omega}{\rightsquigarrow} \langle \mu', e'\rangle}{\langle \mu, E[e]\rangle \overset{\omega}{\rightsquigarrow} \langle \mu', E[e']\rangle}$$

[LIFT]
$$\frac{e \to e'}{\langle t, e\rangle \overset{\epsilon}{\twoheadrightarrow} \langle t, e'\rangle}$$

[LIFT]
$$\frac{e \to e'}{\langle \mu, e\rangle \overset{\epsilon}{\rightsquigarrow} \langle \mu, e'\rangle}$$

[BETA]
$$\frac{}{(\lambda x.e)\ v \to e[v/x]}$$

[RETRY]
$$\frac{v \in I}{\langle e, \mu, \alpha, I, O\rangle \overset{\epsilon}{\Rightarrow} \langle e, \mu, \alpha \uplus \{v \triangleright (e\ v)\}, I, O\rangle}$$

(c) Additional Rules Used To Define the Ideal Semantics of λ_{FAIL}.

[UNIQUE-INPUT]
$$\frac{v \in \langle \mathsf{Val} \rangle, v \notin I}{\langle e, \mu, \alpha, I, O\rangle \overset{in(v)}{\Rightarrow} \langle e, \mu, \alpha \uplus \{v \triangleright (e\ v)\}, I \cup \{v\}, O\rangle}$$

[DUPLINPUT]
$$\frac{v \in \langle \mathsf{Val} \rangle, v \in I}{\langle e, \mu, \alpha, I, O\rangle \overset{in(v)}{\Rightarrow} \langle e, \mu, \alpha, I, O\rangle}$$

[DUPLSEND]
$$\frac{v \in O}{\langle e, \mu, \alpha, I, O\rangle \overset{out(v)}{\Rightarrow} \langle e, \mu, \alpha, I, O\rangle}$$

Figure 3: Operational semantics of λ_{FAIL}.

$$x \in \langle \mathsf{Identifier} \rangle$$

$$\begin{array}{rcl} v \in \langle \mathsf{Val} \rangle & ::= & x \mid \lambda x.e \\ e \in \langle \mathsf{Exp} \rangle & ::= & v \mid e\ e \mid \texttt{atomic}\ v_1\ v_2 \mid \\ & & \texttt{update}\ v_1\ v_2 \mid \texttt{lookup}\ v \end{array}$$

$$\begin{array}{rcl} tn \in \langle \mathsf{TableName} \rangle & = & \langle \mathsf{Val} \rangle \\ t \in \Sigma_{ST} & = & \langle \mathsf{Val} \rangle \mapsto \langle \mathsf{Val} \rangle_{\mathsf{opt}} \\ \mu \in \Sigma_T & = & \langle \mathsf{TableName} \rangle \mapsto \Sigma_{ST} \\ \langle \mathsf{Req} \rangle & = & \langle \mathsf{Val} \rangle \\ \langle \mathsf{Resp} \rangle & = & \langle \mathsf{Req} \rangle \times \langle \mathsf{Val} \rangle \\ (v \triangleright e) \in \Sigma_A & = & \langle \mathsf{Req} \rangle \times \langle \mathsf{Exp} \rangle \\ \Sigma & = & \langle \mathsf{Exp} \rangle \times \Sigma_T \times \Sigma_A^{+} \times 2^{\langle \mathsf{Req} \rangle} \times 2^{\langle \mathsf{Resp} \rangle} \end{array}$$

Figure 2: The Syntax of λ_{FAIL} and its Semantic Domains.

$\langle \mathsf{Val} \rangle_{\mathsf{opt}}$) represents the value of a single table. An element $\mu \in \Sigma_T$ represents the values of all tables.

As explained earlier, an agent represents a thread of computation spawned to process a given input request. The state of an agent is represented by a pair of the form $v \triangleright e$, where v represents the input request being processed and e represents a partially evaluated expression (and represents the local state of the agent). Let Σ_A represent the set of all agent-states.

The state of an executing program is represented by a *system configuration* $\langle e, \mu, \alpha, I, O\rangle \in \Sigma$, where e is the program itself, μ represents the values of all tables, α is the multi-set of currently executing agents, I represents the set of all input requests received so far, and O represents the set of all responses produced so far. (A response to a request v_i is a pair of the form (v_i, v_o) where v_o is the result.) Let Σ represent the set of all configurations.

Let $\uplus$ denote the union operator for multi-sets.

Semantics. Fig. 3 presents an operational semantics for λ_{FAIL} as a labelled transition relation $\overset{r}{\Rightarrow}$ on the set of configurations Σ. The evaluation of a program p starts in the initial configuration $\langle\langle p \rangle\rangle \triangleq \langle p, \mu^I, \emptyset, \emptyset, \emptyset\rangle$, where $\mu^I = \lambda t.\lambda k.\texttt{NONE}$. (Initially, all tables map every key to a default value of `NONE`. We utilize a standard encoding of optional values consisting of either `NONE` or `SOME` v.)

System Transitions. ($\overset{r}{\Rightarrow}$) As rule INPUT indicates, the arrival of an input request v spawns a new agent to evaluate $e\ v$. As rule OUTPUT indicates, when an agent's evaluation completes, the resulting value is sent back as a response. The labels on system transitions represent requests and responses. Rule NORMAL describes a normal system transition caused by a potentially effectful execution step performed by a single agent, described below. As the rule indicates, the execution steps of different agents can be interleaved in a non-deterministic order. Rule FAIL indicates that an agent can fail at any point in time.

Agent Transitions. ($\overset{\omega}{\rightsquigarrow}$) Execution steps in the evaluation of a single agent are described by the transition relation $\overset{\omega}{\rightsquigarrow}$ on $\Sigma_T \times \langle \mathsf{Exp} \rangle$. A transition $\langle \mu, e \rangle \overset{\omega}{\rightsquigarrow} \langle \mu', e' \rangle$ indicates that an agent expression e is transformed to an agent expression e', with the side-effect of transforming the table-state from μ to μ'. The label ω represents a sequence of updates to a single table performed atomically in this step. (This label, however, identifies an internal transition not visible externally, which is why the label is omitted in the corresponding system transition in rule NORMAL.) These transitions are of two types: pure (standard λ-calculus evaluation, BETA), and effectful, which take the form of atomic table operations.

Atomic Table Operations. The expression "`atomic` t e" identifies a set of operations to be performed on a *single* table t in an atomic and failfree fashion. Its semantics is defined by rule ATOMIC, which utilizes a transition relation $\overset{u}{\twoheadrightarrow}$ on atomic evaluation configurations of the form $\langle t, e \rangle$, which indicates the atomic evaluation of an expression e at a table t. The labels on such transitions are either ϵ or represent a single update to a table. Rules UPDATE/LOOKUP define the semantics of an update/lookup operation on a table. The rule ATOMIC indicates that no other execution step interleaves with the evaluation of an atomic expression. Note that the evaluation of an atomic expression cannot fail in the middle. In other words, either all effects of the atomic expression evaluation happen or none does.

Retry. The rule RETRY models one of the key ingredients used to tolerate process failures, namely a retry mechanism. The rule indicates that a request must be *retried* periodically. Typically, retrying logic is built into clients. A client will resend a request if it does not receive a response within a pre-defined amount of time. This basic scheme can be optimized, as discussed in Section 4: the system (application) can send an acknowledgement back to the client, after which the client can stop resending a request and the application takes on the responsibility of retrying the request (to ensure progress in the presence of failures). The system can exploit various optimizations in implementing the retry logic, but rule RETRY suffices for our purpose here. As we will soon see, the key reason for adding the RETRY rule to the semantics is to formalize a weakened progress guarantee ("progress modulo retries") that is appropriate in the presence of failures.

2.2 Formalizing Failfree Idempotence

We now formalize a natural correctness goal of any λ_{FAIL} program: namely, that it correctly handles process failures and duplicate messages. We will later see how we can automatically ensure this property for any program. We formalize this correctness criterion as follows. We define an alternative semantics for λ_{FAIL}, which we refer to as the *ideal semantics*, representing an idealized execution platform. We then define a program to be "correct" iff its behavior under the standard semantics is equivalent to its behavior under the ideal semantics.

Ideal Semantics. Let A denote the set of all rules defined in Fig. 3(b). We will define the ideal semantics as a different labelled transition relation on program configurations, by adding and removing some rules to set A. Let $\Rightarrow_S$ denote the labelled transition relation on Σ induced by a given set of rules S. Thus, $\Rightarrow_{\mathsf{A}}$ is the transition relation capturing the standard semantics of λ_{FAIL}.

We first omit rule FAIL eliminating process failures from the ideal semantics. In the ideal semantics, we assume that all input requests are distinct, by replacing the INPUT rule by the UNIQUE-INPUT and DUPLINPUT rules. We also drop rule RETRY. Finally, process failures make it hard to ensure that an application sends an output message exactly once. A common approach to this problem is to weaken the specification and permit the application to send the same output message more than once. We do this by adding rule DUPLSEND. We define IDEAL to be $\mathsf{A} \setminus \{$ FAIL, RETRY, INPUT$\} \cup \{$ UNIQUE-INPUT, DUPLINPUT, DUPLSEND$\}$. We refer to $\Rightarrow_{\mathsf{IDEAL}}$ as the ideal semantics for λ_{FAIL}.

Observational Idempotence (Safety). We now consider two (well-studied) notions of behavioral equivalence in formalizing our correctness criterion. Recall that $\langle\!\langle p \rangle\!\rangle$ denotes the initial configuration in an execution of program p (which is the same under both semantics). Given a labelled transition relation $\Rightarrow$ on configurations, an *execution* of a program p (with respect to $\Rightarrow$) is an alternating sequence of states and labels, denoted $\sigma_0 \overset{r_1}{\Rightarrow} \sigma_1 \cdots \sigma_n$, representing a sequence of transitions starting from the initial program state $\sigma_0 = \langle\!\langle p \rangle\!\rangle$. We say that the *observed behavior* $obs(\xi)$ of an execution ξ is the sequence of non-ϵ labels in ξ. Note that $obs(\xi)$ is a sequence of (input) requests and (output) responses. Specifically, it does not include updates to tables, which are internal transitions but not visible externally.

DEFINITION 2.1. *We say that a λ_{FAIL} program p is* observationally idempotent *if for every execution ξ_1 of p under the standard semantics there exists an execution ξ_2 of p under the ideal semantics such that $obs(\xi_1) = obs(\xi_2)$.*

We present a simpler, more abstract, formalization of idempotence in the appendix. However, the preceding definition in terms of the ideal semantics will be useful in formalizing progress properties, as below, and in proving correctness of our implementation.

Failfree Idempotence (Progress). An observationally idempotent program, by definition, gives no progress guarantees. Consider a modified version of the account-transfer example that checks the input request to determine if it is a duplicate request and processes it only if it is not a duplicate. This ensures that the program is idempotent. However, if the agent fails in between the debit and credit steps, we would still have a problem. This motivates the following stronger correctness condition, based on the notion of weak bisimulation. A labelled transition system $(S, \Rightarrow)$ consists of a relation $\overset{\ell}{\Rightarrow} \subseteq S \times S$ for every $\ell \in \langle \mathsf{Label} \rangle$.

DEFINITION 2.2. *A weak bisimulation between two labelled transition systems $(\Sigma_1, \Rightarrow_1)$ and $(\Sigma_2, \Rightarrow_2)$ is a relation $\sim \subseteq \Sigma_1 \times \Sigma_2$ such that for any $\sigma_1 \sim \sigma_2$ we have:*

1. $\sigma_1 \overset{\ell}{\Rightarrow}_1 \sigma_1' \wedge \ell \neq \epsilon \;\Rightarrow\; \exists \sigma_2', \sigma_2''.\sigma_2 \overset{\epsilon}{\Rightarrow}_2{}^{*} \sigma_2'' \overset{\ell}{\Rightarrow}_2 \sigma_2' \wedge \sigma_1' \sim \sigma_2'$
2. $\sigma_2 \overset{\ell}{\Rightarrow}_2 \sigma_2' \wedge \ell \neq \epsilon \;\Rightarrow\; \exists \sigma_1', \sigma_1''.\sigma_1 \overset{\epsilon}{\Rightarrow}_1{}^{*} \sigma_1'' \overset{\ell}{\Rightarrow}_1 \sigma_1' \wedge \sigma_1' \sim \sigma_2'$
3. $\sigma_1 \overset{\epsilon}{\Rightarrow}_1 \sigma_1' \;\Rightarrow\; \exists \sigma_2'.\sigma_2 \overset{\epsilon}{\Rightarrow}_2{}^{*} \sigma_2' \wedge \sigma_1' \sim \sigma_2'$
4. $\sigma_2 \overset{\epsilon}{\Rightarrow}_2 \sigma_2' \;\Rightarrow\; \exists \sigma_1', \sigma_1'.\sigma_1 \overset{\epsilon}{\Rightarrow}_1{}^{*} \sigma_1' \wedge \sigma_1' \sim \sigma_2'$

We will write $(\sigma_1, \Sigma_1, \Rightarrow_1) \simeq (\sigma_2, \Sigma_2, \Rightarrow_2)$ to indicate that there exists a weak bisimulation $\sim$ between $(\Sigma_1^r, \Rightarrow_1)$ and $(\Sigma_2^r, \Rightarrow_2)$ under which $\sigma_1 \sim \sigma_2$, where Σ_i^r represents the set of states in Σ_i that are reachable from σ_i via a sequence of $\Rightarrow_i$ transitions. We will omit Σ_1 and Σ_2 in this notation if no confusion is likely.

DEFINITION 2.3. *A λ_{FAIL} program p is said to be* failfree idempotent *iff $(\langle\!\langle p \rangle\!\rangle, \Rightarrow_{\mathsf{A}}) \simeq (\langle\!\langle p \rangle\!\rangle, \Rightarrow_{\mathsf{IDEAL}})$.*

This definition requires a failfree idempotent program to provide progress guarantees: at any point in time, if the system can produce a response r under the ideal semantics, then the system should be capable of producing the same response r under the standard semantics also. However, the inclusion of rule RETRY in the standard semantics means that this progress guarantee holds *provided requests are retried*. Absolute progress guarantees are not possible since an agent may fail before it executes even its first step.

THEOREM 2.4. *A failfree idempotent program is also observationally idempotent.*

```
let imreturn v = fun (guid, tc) → (tc, v)

let imatomic T f =
  fun (guid, tc) →
    atomic T {
      let key = (0, (guid, tc)) in
      match lookup key with
      | Some(v) → (v,tc+1)
      | None →
        let v = f () in (update key v); (v,tc+1)
    }

let imbind (idf, f) =
  fun (guid, tc) →
    let (ntc, v) = idf (guid, tc) in
    f v (guid, ntc)

let imrun idf x =
  let (j,v) = idf x (x, 0) in v

let imupdate key val = update (1,key) val
let imlookup key = lookup (1,key)
```

Figure 4: The Idempotence Monad.

Failfree Idempotent Realization. Failfree idempotence is a generic correctness property we expect of a λ_{FAIL} program. More generally, the following definition combines this property with a "specification" (provided as another λ_{FAIL} program q).

DEFINITION 2.5. *A λ_{FAIL} program p is said to be a* failfree idempotent *realization of q iff $(\langle\langle p \rangle\rangle, \Rightarrow_{\mathsf{A}}) \simeq (\langle\langle q \rangle\rangle, \Rightarrow_{\mathsf{IDEAL}})$.*

3. Realizing Failfree Idempotence

We now present a generic, application-independent, strategy that can be used to ensure failfree idempotence. Informally, a function is idempotent if multiple evaluations of the function on the same argument, potentially concurrently, behave the same as a single evaluation of that function with the same argument. Consider the example in Fig. 1. In this example, the parameter *requestId* serves to distinguish between different transfer requests (and identify duplicate requests). This service can be made idempotent and failfree by (a) using this identifier to *log* debit and credit operations, whenever the operations are performed, and (b) modifying the debit and credit steps to check, using the log, if the steps have already been performed. This strategy can ensure that multiple (potentially partial and concurrent) invocations of *transfer* with the same identifier have the same effect as a single invocation.

Manually ensuring idempotence is tedious and it introduces the possibility of various subtle bugs and in general makes implementation less comprehensible. We now describe a monad-based library that realizes idempotence and failure-freedom in a generic way.

3.1 The Idempotence Monad

The Intuition. A computation performed by a single agent consists of a sequence of steps, "pure" as well as "effectful" ones (namely, atomic local transactions which can read/update a single table.) We use the following strategy to make a computation idempotent:

1. Associate every computation instance (that we wish to make idempotent) with a unique identifier.
2. Associate every step in an idempotent computation with a unique number.
3. Whenever an effectful step is executed, we simultaneously persistently record the fact that this step has executed and save the value produced by this step.
4. Every effectful step is modified to first check if this step has already been executed. If it has, then the previously saved value (for this step) is used instead of executing the step again.

Note that λ_{FAIL} does not have any non-deterministic construct (in a single agent's evaluation). Non-deterministic constructs can be supported by treating them as an effectful step so that once a non-deterministic choice is made, any re-execution of the same step makes the same choice.

Details. We now describe in detail how individual computation steps can be made idempotent and how these idempotent steps can be composed together into idempotent computation. Our solution is essentially a monad (Fig. 4).

We represent an idempotent computation as a function that takes a tuple $(guid, tc)$ as a parameter (where $guid$ and tc represent an identifier and a step number used to uniquely identify steps in a computation) and returns a value along with a new step number. We can "execute" an idempotent computation *idf* as shown by the function `imrun`, using the function's argument itself as the $guid$ value and an initial step number of 0.

The function `imreturn` (the monadic "return") lifts primitive values to idempotent computations. This transformation can be used for any pure (side-effect-free) expressions.

Side-effects (table operations) are permitted only inside the `atomic` construct. The function `imatomic` is used to make a local transaction idempotent. Specifically, "`imatomic` T f_m" is an idempotent representation for "`atomic` T f", where f_m is the monadic form of f constructed as described later. As explained above, this is represented as a function that takes a pair $(guid, tc)$ as a parameter and realizes the memoization strategy described above. The pair $(guid, tc)$ is used as a unique identifier for this step. We check whether this step has already executed. If so, we return the previously computed value. If not, we execute this computation step and memoize the computed value. In either case, the step number is incremented in this process and returned.

It is, however, critical to do all of the above steps *atomically* to ensure that even if two agents concurrently attempt to execute the same computation, only one of them will actually execute it. However, note that an atomic expression is allowed to access only a single table. Hence, the "memoized" information for this computation step must be stored in the same table that is accessed by the computation step. However, we must keep our memoization and book-keeping information logically distinct from the program's own data stored in the same table. We achieve this by creating two distinct "address spaces" (of key values) for the table. We convert a key value k used by our idempotence implementation to $(0, k)$ and convert a key value k used by the program itself to $(1, k)$. The functions `imupdate` and `imlookup` do this wrapping for the user's code. Thus, the expression f to be evaluated in the atomic transaction is transformed to its monadic representation f_m by replacing `lookup` and `update` in f by `imlookup` and `imupdate` respectively.

We now consider how to compose individual computation steps in an idempotent computation (the monadic "*bind*" function). Consider a computation of the form "$let\ x\ =\ step1\ in\ step2$" which is equivalent to "$(\lambda x.step2)step1$". We transform $step1$ to its monadic form, say *idf*. We transform $step2$ to its idempotent form, say g. The function *bind*, applied to *idf* and $\lambda x.g$, produces the idempotent form of the whole computation. (A formal and more complete description of the transformation is presented later.)

The result of "`imbind` *idf f*" is defined as follows: it is an idempotent function that takes a parameter $(guid, tc)$, and invokes

idf (the first step). It uses the value and the step number returned by the idempotent function and invoke the second idempotent function f. Thus the monad effectively threads the step number through the computation, incrementing it in every atomic transaction.

Note that a key characteristic of this implementation is that it is completely decentralized. When a transaction completes, the current agent simply attempts to execute the next (idempotent) transaction in the workflow; no coordination with a centralized transaction manager is required. In general, distributed coordination requires the use of expensive blocking protocols such as 2 phase commit. In contrast, a workflow implementation based on this idea of idempotence coupled with (client-side) retry logic are non-blocking. The implementation does not have a single point-of-failure or single performance bottleneck, which can lead to increased scalability.

Also note that this implementation creates one log entry per transaction. Once logging for idempotence is separated from the core business logic, it can be optimized in several ways. We can avoid redundant logging if the underlying storage engine maintains its own log. Logging can be avoided if the transaction is (declared to be) semantically idempotent.

Example. The monadic library lets us construct the monadic version e_m of any λ_{FAIL} expression e, by using the monadic version of any primitive and the monadic bind in place of function application. E.g., the monadic version of the following code fragment (from the money transfer example)

atomic fb {*update fa* ((*lookup fa*) − *amt*)};
atomic tb {*update ta* ((*lookup ta*) + *amt*)};

is the following:

```
imbind
  (imatomic fb {imupdate fa ((imlookup fa) − amt)})
  {imatomic tb {imupdate ta ((imlookup ta) + amt)}}
```

where $\{e\}$ is shorthand for $\lambda x.e$, where x is not free in e.

3.2 Idempotence Monad Programs Are Failfree

In this section, we show that programs written using the idempotence monad are failfree.

Idempotent Executions. Consider any execution of a λ_{FAIL} program. We refer to any transition generated by rule NORMAL as an *execution-step* and identify it by the triple $(v \triangleright e) \xrightarrow{\omega} (v \triangleright e')$. Thus, an *execution-step* $a \xrightarrow{\omega} b$ represents a transition caused by an agent a that transforms to an agent b, with a side-effect ω on the tables. An execution-step $a \xrightarrow{\omega} b$, after an execution ξ, is said to be a *repeated* execution-step if the execution ξ contains some execution-step of the form $a \xrightarrow{\omega'} b'$. (Note that this does not signify a cycle in the execution since the states of the tables could be different in the two corresponding configurations.)

DEFINITION 3.1. *An execution is said to be* operationally idempotent *iff for any two execution-steps* $a \xrightarrow{\omega} b$ *and* $a' \xrightarrow{\omega'} b'$ *in the execution (in that order),* $a = a'$ *implies that* $b = b'$ *and* ω' *is empty.*

Note that operational idempotence is a property that involves the side-effects on the tables, unlike observational idempotence. As we show below, it is a stronger property that can be used to establish observational idempotence.

LEMMA 3.2. *Any operationally idempotent execution* ξ *of a program p (in the standard semantics) is observationally equivalent to some ideal execution* ξ' *of p (in the ideal semantics): i.e.,* $obs(\xi) = obs(\xi')$.

PROOF SKETCH. Let ξ be an operationally idempotent execution of a program p under the standard semantics $\Rightarrow_{\mathsf{A}}$. We show how to construct an execution ξ' of p under the $\Rightarrow_{\mathsf{IDEAL}}$ semantics such that $obs(\xi) = obs(\xi')$.

We first omit any transitions in ξ due to the FAIL rule since their labels are empty.

We next omit the transitions corresponding to repeated execution-steps:

The key point to note here is that an execution-step $a \xrightarrow{\omega} b$ affects the (legality of) subsequent transitions in two ways: indirectly through the side-effects ω on the tables and directly through b (which may take part in subsequent transitions). A repeated execution-step is redundant in any idempotent execution that has no failures (transitions due to the FAIL rule): it has no side-effects on the tables, and if the value b is subsequently used in a non-repeated execution-step, then we can show that another agent identical to b already exists in the configuration.

We then omit any transition due to the RETRY rule. We finally replace INPUT transitions corresponding to a duplicate and replace INPUT transitions corresponding to a non-duplicate input request by a UNIQUE-INPUT transition.

This leaves us with duplicate responses that may be produced by the OUTPUT rule for the same input (due to multiple agents that process it). We replace these transitions by corresponding DUPLSEND transition.

This transformation produces an execution ξ' of p under the $\Rightarrow_{\mathsf{IDEAL}}$ semantics such that $obs(\xi) = obs(\xi')$. ■

Our next goal is to show that all executions of programs written using the idempotence monad are operationally idempotent. However, this claim requires the programs to be *well-typed*, as defined below.

Well-Typed IM Programs. The idempotence monad can be used to write monadic programs in the usual way. The monad may be thought of as defining a new language λ_{IM}, obtained from λ_{FAIL} by replacing the keywords `atomic`, `lookup`, and `update` by the keywords `imatomic`, `imlookup`, `imupdate`, `imbind`, and `imrun`. A λ_{IM} program can also be thought as a λ_{FAIL} program, using the definition of the monad in Fig. 4.

We first mention a few type restrictions used to simplify presentation. We refer to λ-calculus terms as *pure* and their types as pure types. We assume that the types of keys and values of all tables are pure types. We assume that the types of the input request and the output response are also pure types.

We use $\langle \tau_k, \tau_v \rangle$ to denote the type of a table with keys of type τ_k and values of type τ_v. We assume that table names come from some fixed set of identifiers and that the typing environment maps these table names to their types.

The non-pure terms can be classified into two kinds. The first kind are expressions (such as `imupdate` k v), which are meant to be evaluated in the context of a single table (as part of a local transaction). Fig. 5 presents typing rules for intra-transaction expressions. The type $\tau!\langle \tau_k, \tau_v \rangle$ signifies a term that can be evaluated in the context of a table of type $\langle \tau_k, \tau_v \rangle$, producing a value of type τ. The second kind of expressions are the ones that use the idempotence monad, used to represent workflow computations which execute one or more local transactions. Fig. 6 presents typing rules for such expressions (which are the standard monad typing rules).

In the sequel, we will use the term well-typed IM program to refer to any expression e such that $\Gamma_{tn} \models e : \tau_i \rightarrow_{\text{ID}} \tau_o$, where τ_i and τ_o are, respectively, the types of input and output messages, and Γ_{tn} provides the typings of tables. This is essentialy a program of the form `imrun` e', where e' is constructed from the other monadic constructs.

$$\text{[IMLOOKUP]}\quad \frac{\Gamma \models e : \tau_k!\langle \tau_k, \tau_v\rangle}{\Gamma \models \texttt{imlookup}\, e : \tau_v!\langle \tau_k, \tau_v\rangle} \qquad \text{[IMUPDATE]}\quad \frac{\Gamma \models e_1 : \tau_k!\langle \tau_k, \tau_v\rangle \qquad \Gamma \models e_2 : \tau_v!\langle \tau_k, \tau_v\rangle}{\Gamma \models \texttt{imupdate}\, e_1\, e_2 : \text{unit}!\langle \tau_k, \tau_v\rangle} \qquad \text{[AT-VAR]}\quad \frac{}{x : \tau \models x : \tau!\langle \tau_k, \tau_v\rangle}$$

$$\text{[AT-LAMBDA]}\quad \frac{\Gamma, x : \tau_1 \models e : \tau_2!\langle \tau_k, \tau_v\rangle}{\Gamma \models \lambda x.e : (\tau_1 \overset{\langle \tau_k, \tau_v\rangle}{\to} \tau_2)!\langle \tau_k, \tau_v\rangle)} \qquad \text{[AT-APPLY]}\quad \frac{\Gamma \models e_1 : (\tau_1 \overset{\langle \tau_k, \tau_v\rangle}{\to} \tau_2)!\langle \tau_k, \tau_v\rangle \qquad \Gamma \models e_2 : \tau_1!\langle \tau_k, \tau_v\rangle}{\Gamma \models e_1 e_2 : \tau_2!\langle \tau_k, \tau_v\rangle}$$

Figure 5: A type system for intra-transaction expressions.

$$\text{[IMRETURN]}\quad \frac{\Gamma \models e : \tau}{\Gamma \models \texttt{imreturn}\, e : \mathsf{IM}\ \tau} \qquad \text{[IMATOMIC]}\quad \frac{\Gamma \models t : \langle \tau_k, \tau_v\rangle \qquad \Gamma \models e : (\text{unit} \overset{\langle \tau_k, \tau_v\rangle}{\to} \tau)!\langle \tau_k, \tau_v\rangle}{\Gamma \models \texttt{imatomic}\, t\, e : \mathsf{IM}\ \tau}$$

$$\text{[IMBIND]}\quad \frac{\Gamma \models e_2 : \tau_1 \to \mathsf{IM}\ \tau_2 \qquad \Gamma \models e_1 : \mathsf{IM}\ \tau_1}{\Gamma \models \texttt{imbind}\, e_1\, e_2 : \mathsf{IM}\ \tau_2} \qquad \text{[IMRUN]}\quad \frac{\Gamma \models e : \tau_1 \to \mathsf{IM}\ \tau_2}{\Gamma \models \texttt{imrun}\, e : \tau_1 \to_{\text{ID}} \tau_2}$$

$$\text{[VAR]}\quad \frac{}{x : \tau \models x : \tau} \qquad \text{[LAMBDA]}\quad \frac{\Gamma, x : \tau_1 \models e : \tau_2}{\Gamma \models \lambda x.e : \tau_1 \to \tau_2} \qquad \text{[APPLY]}\quad \frac{\Gamma, x : \tau_1 \models e : \tau_2}{\Gamma \models \lambda x.e : \tau_1 \to \tau_2}$$

Figure 6: A type system for idempotence monad.

LEMMA 3.3. *All executions of a well-typed IM program are operationally idempotent.*

PROOF SKETCH. Let us refer to a value of the form $(0, k)$ (used as a key for a table lookup/update operation) as a *system key*. Consider any execution of a well-typed IM program. It is easy to verify that once the value associated with a system key $(0, k)$ in any table is set to be v, it is never subsequently modified.

Consider any two execution-steps $a \overset{\omega}{\dashrightarrow} b$ and $a' \overset{\omega'}{\dashrightarrow} b'$ in the execution (in that order) where $a = a'$. The idempotence property follows trivially whenever these steps are effect-free steps. The only effectful steps are produced by the evaluation of an `imatomic` construct. Since $a = a'$, the *key* value used in the evaluation of `imatomic` in both steps must be identical. The implementation of `imatomic` guarantees that whatever value is produced by the evaluation of `imatomic` in the first step will be memoized with the given value. This guarantees that the second step will find this same value in the table (thanks to the property described earlier). Hence, it will evaluate to the same value and will have no side-effects on the table. It follows that the execution is idempotent. ■

THEOREM 3.4. *Any well-typed IM program is observationally idempotent.*

PROOF SKETCH. Follows immediately from the previous two lemmas. ■

As stated earlier, observational idempotence does not give us any progress guarantees in the presence of failures. We now establish progress guarantees in the form of a weak bisimulation.

DEFINITION 3.5. *Let p be any λ_{FAIL} program. Let Σ^p_{A} denote the set of configurations $\{\ \sigma \in \Sigma \mid \langle\langle p\rangle\rangle \Rightarrow^*_{\text{A}} \sigma\ \}$ that can be produced by the execution of p under the standard semantics. Let Σ^p_{I} denote the set of configurations $\{\ \sigma \in \Sigma \mid \langle\langle p\rangle\rangle \Rightarrow^*_{\mathsf{IDEAL}} \sigma\ \}$ that can be produced by the execution of p under the ideal semantics. We define the relation $\simeq$ between Σ^p_{A} and Σ^p_{I} by $\langle p, \mu_1, \alpha_1, I_1, O_1\rangle \simeq \langle p, \mu_2, \alpha_2, I_2, O_2\rangle$ iff $I_1 = I_2$ and $\mu_1 = \mu_2$.*

Note that in the above definition, the condition $I_1 = I_2$ implies that both configurations have received the same set of input requests. The condition $\mu_1 = \mu_2$ implies that the processing of each input request v in both configurations have gone through identical sequences of effectful steps so far. (This follows since every effectful step is memoized and is part of the table state μ_1 and μ_2.)

THEOREM 3.6. *If p is a well-typed IM program, then $\simeq$ is a weak bisimulation between $(\Sigma^p_{\text{A}}, \Rightarrow_{\text{A}})$ and $(\Sigma^p_{\text{I}}, \Rightarrow_{\mathsf{IDEAL}})$.*

PROOF SKETCH. Consider any $\sigma_1 \simeq \sigma_2$. The interesting transitions are those that are effectful (involving an atomic operation) or produce an output. If either σ_1 or σ_2 can perform an interesting transition, we must show that the other can perform an equivalent transition (possibly after a sequence of silent transitions).

Consider the case when an agent $v \triangleright e$ in σ_1 can perform an effectful transition in the standard semantics. Let $v \triangleright e_0$ be the state of the same agent after its most recent `imatomic` evaluation. Then, we must have some side-effect-free evaluation $e_0 \to e_1 \cdots e_k = e$. Consider configuration σ_2. By definition, we restrict our attention to reachable states. Hence, there exists some execution in the ideal semantics that produces σ_2. This execution must have received input request v and produced an agent $v \triangleright (p\ v)$. Consider the evaluation of this agent. The effectful steps in this agent's evaluation in the ideal semantics must have produced the same sequence of values as in the evaluation in the standard semantics (since the memoized values for these steps are the same in both σ_1 and σ_2). Thus, σ_2 must have some agent of the form $v \triangleright e_i$ for some $0 \le i \le k$. This will produce, after zero or more silent transitions, the agent $v \triangleright e$ that can then perform the same effectful transition in the ideal semantics as in the standard semantics.

Consider the case when an agent $v \triangleright e$ in σ_2 can perform an effectful transition in the ideal semantics. We can create a new agent $v \triangleright (p\ v)$ using rule RETRY in the standard semantics. We can then

$$
\begin{array}{rcll}
[x]^{\mathsf{V}} & = & x & \\
[\lambda x.e]^{\mathsf{V}} & = & \lambda x.[e]^{\mathsf{I}} &
\end{array}
$$

$$
\begin{array}{rcll}
[x]^{\mathsf{I}} & = & \texttt{imreturn}\ x & \\
[\lambda x.e]^{\mathsf{I}} & = & \texttt{imreturn}\ (\lambda x.[e]^{\mathsf{I}}) & \\
[n\ e]^{\mathsf{I}} & = & \texttt{imbind}\ [n]^{\mathsf{I}}\ (\lambda x.[x\ e]^{\mathsf{I}}) & ,n \notin \langle \mathsf{Val} \rangle \\
[v\ n]^{\mathsf{I}} & = & \texttt{imbind}\ [n]^{\mathsf{I}}\ (\lambda x.[v\ x]^{\mathsf{I}}) & ,v \in \langle \mathsf{Val} \rangle, n \notin \langle \mathsf{Val} \rangle \\
[v\ v']^{\mathsf{I}} & = & [v]^{\mathsf{V}}\ [v']^{\mathsf{V}} & ,v, v' \in \langle \mathsf{Val} \rangle \\
[\texttt{atomic}\ x\ e]^{\mathsf{I}} & = & \langle \texttt{imatomic}\ x\ [e]^{\mathsf{A}}, fv \rangle & \\
 & & \text{where } fv = \{x\} \cup \mathit{freevars}(e) &
\end{array}
$$

$$
\begin{array}{rcl}
[x]^{\mathsf{A}} & = & x \\
[\lambda x.e]^{\mathsf{A}} & = & \lambda x.[e]^{\mathsf{A}} \\
[e_1 e_2]^{\mathsf{A}} & = & [e_1]^{\mathsf{A}}[e_2]^{\mathsf{A}} \\
[\texttt{update}\ e_1\ e_2]^{\mathsf{A}} & = & \texttt{imupdate}\ [e_1]^{\mathsf{A}}\ [e_2]^{\mathsf{A}} \\
[\texttt{lookup}\ e]^{\mathsf{A}} & = & \texttt{imlookup}\ [e]^{\mathsf{A}}
\end{array}
$$

$$
\begin{array}{rcl}
\langle e, \{\} \rangle & = & e \\
\langle e, \{x\} \uplus Y \rangle & = & \langle \texttt{imbind}\ x\ (\lambda x.e), Y \rangle
\end{array}
$$

Figure 7: Transforming λ_{FAIL} expressions into idempotent expressions.

duplicate the entire execution history of $v \triangleright e$ (which is guaranteed to be the same in both semantics by the definition of $\simeq$). Thanks to the idempotence property, this duplicate execution will have no extra side-effects and will eventually produce the same effectful transition as in the ideal semantics. ■

THEOREM 3.7. *A well-typed IM program is failfree.*

3.3 Transforming λ_{FAIL} Programs To Idempotent Programs

We have seen that the idempotence monad lets us construct failfree programs. Given any λ_{FAIL}-program e, we can construct its "equivalent" monadic representation using standard techniques (which are conceptually straightforward, though the details are intricate). The transformation is presented in Fig. 7 (based on the transformation in [17]).

We make a few simplifying assumptions (type restrictions) about the source λ_{FAIL} program e in this transformation algorithm (similar to those mentioned in Section 3.2). We assume that the types of keys and values of all tables are pure types. We assume that intra-transaction expressions do not manipulate values of workflow type: e.g., `atomic` t $\{\lambda x.(\texttt{atomic}\ s\ e)\}$ is not allowed.

The translation is defined using multiple translation functions. $[v]^{\mathsf{V}}$ is a translation function that applies only to values (which must be of the form x or $\lambda x.e$). The translation function $[e]^{\mathsf{I}}$ is the heart of the monadic transformation and can be applied to multi-transaction expressions. It uses the idempotence monad to sequence local transactions. The translation function $[e]^{\mathsf{A}}$ is applied to intra-transaction expressions and it is used primarily to replace occurrences of `update` and `lookup` by `imupdate` and `imlookup` respectively. Note that a single local transaction (i.e.intra-transaction expression) evaluation is done using standard evaluation (without using any monad). The auxiliary function $\langle e, X \rangle$ is used to transform the monadic values used in the evaluation of multi-transactions to standard values required in the evaluation of a single local transaction.

Finally, given a top-level λ_{FAIL} program e (which is assumed to be a functional value of the form $\lambda x.e'$), it's monadic form e_m is defined to be `imrun` $[e]^{\mathsf{V}}$. As usual, it can be shown that the translation preserves types: translation of a well-typed λ_{FAIL} program (satisfying the restrictions mentioned above) will produced a well-typed monadic program.

3.4 Failfree Realization Via Idempotence

Given any λ_{FAIL} expression e, we can construct its monadic version e_m as explained above. The preceding results imply that e_m must be failfree. We now establish a stronger result, namely that e_m is a failfree realization of e. (Failure-freedom is a weak correctness criterion. It indicates that a program's behavior under the standard semantics is equivalent to *it's* behavior under the ideal semantics.) The notion of failure-freedom does let us simplify verifying correctness of a program by considering only its behavior under the ideal semantics. In particular, we have:

THEOREM 3.8. *If p and q are weakly bisimilar under the ideal semantics (i.e., if $(\langle\langle p \rangle\rangle, \Rightarrow_{\mathsf{IDEAL}}) \simeq (\langle\langle q \rangle\rangle, \Rightarrow_{\mathsf{IDEAL}})$, and p is failfree, then p is a failfree realization of q.*

PROOF SKETCH. Follows as we can compose the two weak bisimulations together. ■

This theorem simplifies proving that a program is a failfree realization of another. In particular, we have already seen that a monadic program is failfree (Theorem 3.7). Hence, to prove that e_m is a failfree realization of e, it suffices to show a weak bisimilarity between the *ideal* executions of a λ_{FAIL} program e and the *ideal* executions of its monadic representation e_m.

THEOREM 3.9. *Let e_m be the monadic representation of e. Then, e_m is a failfree realization of e.*

PROOF SKETCH. As explained above, it is sufficient to relate evaluations of e and e_m under the ideal semantics. In this setting, it is intuitively simple to see why the monadic program "e_m" simulates the given program e. The key distinction is that the monadic implementation uses `imatomic` to perform any effectful step. This step will first check the memoized data to see if this step has already executed. In an ideal execution, *this check will always fail*, and the monadic implementation then performs the same effectful step as the original program (and memoizes it). The check is guaranteed to always fail because the keys (g, i) used in distinct executions of `imatomic` are distinct. The value of g will be different for executions corresponding to different inputs x and y. The value i will be different for different steps corresponding to the same input x. ■

3.5 Idempotent Failfree Computations as a Language Feature

We have now seen how idempotent failfree computations can be automatically realized using the idempotence monad. We now propose a new language construct for idempotent failfree computations.

The construct "`idff` v e" indicates that the computation of e should be failfree and should be idempotent with respect to v: i.e., multiple (potentially concurrent) invocations of this construct with the same value v behaves as though only one of the invocations executed. The above construct permits users to specify the code fragment for which they desire *automatic* idempotence and failure-freedom (provided by the compiler). This enables users to rely on other methods, perhaps application-specific, to ensure these properties elsewhere. (For instance, some computation may be semantically idempotent already.) It also lets the users specify what should be used as *computation identifier* to detect duplicates, as illustrated below.

We refer to this enhanced language as λ_{IDFF}. A formal semantics of λ_{IDFF} appears in the appendix. We illustrate the meaning of

the first parameter of `idff` using the incorrect λ_{FAIL} example of Fig. 1. We can wrap the `idff` construct around this example in the following two ways, with different semantics. Note that the input in this example is a pair (*reqId, req*) consisting of a request-id as well as the actual request. Consider:

$$f_1 = \lambda(reqId, req).\ \texttt{idff}\ reqId\ (process\ (reqId, req))$$
$$f_2 = \lambda(reqId, req).\ \texttt{idff}\ (reqId, req)\ (process\ (reqId, req))$$

The behavior of these two functions differ in the cases where multiple inputs arrive with the same request-id but differing in the second parameter *req*. f_1 treats such requests as the same and will process only one of them, while f_2 will treat them as different requests and process them all.

One of the subtle issues with the semantics and implementation of the `idff` construct is the treatment of invocations that have the same id (first parameter) but have different expressions as the second parameter. We take a simple approach with our semantics (that the effect is as if only the first invocation occurred). This has some implications for the underlying implementation (in the presence of failures). One solution is to use a continuation-passing style computation, and memoize the entire continuation in the memoization step (rather than just the value computed for the step).

4. Extensions

We have seen how idempotence can serve as the basis for *fail-free composition* of computations: essentially, a simple form of *fault-tolerant* workflow. In this section, we describe two extensions that enrich this construct, namely compensating actions and asynchronous evaluation, which simplify writing applications. These concepts are not new, but what is interesting is that they can be integrated without affecting the light-weight, decentralized, nature of our idempotence implementation.

4.1 Compensating Actions

The `idff` construct allows us to compose several transactions into an idempotent workflow that appears to execute *exactly once* without process failures. However, the lack of isolation means that when a transaction in the workflow is executed, its precondition may not be satisfied and we may need to abort the workflow. For example, in the transfer example (Fig. 1), the debit step may succeed, but we may be unable to complete the subsequent credit step because the account does not exist. One way of recovering from this failure is to *compensate for* the debit by crediting the amount back to the source account. If compensating actions are correct, the workflow can guarantee all-or-nothing semantics i.e. either all or none of the transactions in the workflow appear to execute.

We first formalize the desired semantics of workflows with compensating actions. We present a language λ_{IDWF} which provides language constructs to associate transactions with compensating actions, and to declare logical failures. Finally, λ_{IDWF} supports a construct `idworkflow` $id\ e$, which composes transactions with compensating actions into a workflow. We present semantics of this construct, and then show how this construct is realized using a *compensation monad*.

Syntax. λ_{IDWF} modifies and extends λ_{FAIL} in the following ways (see Fig. 8). `atomic` $t\ e_a\ e_c$ extends the `atomic` construct of λ_{FAIL} by specifying e_c as the compensation for the atomic transaction e_a. `abort` indicates that a logical failure has occured and the workflow must be aborted. `idworkflow` $id\ e$ represents an idempotent workflow with identifier id, where e is the workflow consisting of a composition of atomic transactions with compensations. Expression of the form $e \oslash e_c$ arise only during evaluation and are not source language constructs.

Semantic Domains. The semantic domains for λ_{IDWF} are the same as for λ_{FAIL} with minor extensions. The runtime expression $e \oslash e_c$ is used to represent a workflow during its execution. Here, e represents the partially evaluated form of a workflow and e_c represents the compensating action to be performed in case the workflow needs to be aborted. Agents can also be of the form $id \blacktriangleright e_w \oslash e_c$, indicating an agent evaluating a workflow.

Semantics. The semantics of λ_{IDWF} is defined using a set of rules consisting of all the rules in A used to define the semantics of λ_{FAIL} except for FAIL and ATOMIC, plus the new rules presented in Fig. 8.

The initiation of a workflow (rule IDWF-BEGIN) creates a new agent of the form $id \blacktriangleright e_w \oslash e_c$, provided no agent has already been created for id. This agent evaluates the workflow e_w and tracks the compensating action e_c to be performed in case of an `abort`. Rule IDWF-END indicates how the computation proceeds once the workflow evaluation is complete (or if a previous workflow with the same id has already been initiated).

The rules ATOMIC and ATOMIC-ABORT define the semantics of transactions in a workflow. Informally, the expression `atomic` $t\ e_a\ e_c$ is evaluated as follows. First, e_a is evaluated atomically to produce a value v. Then, as a side-effect, the compensating action is updated to indicate that $e_c\ v$ should be evaluated as the first step of the compensation before executing the original compensation. Finally, the whole expression evaluates to v. Thus, note that the value produced by the "atomic" action e_a is available to the subsequent computation as well as the compensating action e_c. When a workflow is aborted (rules ATOMIC-ABORT and IDWF-ABORT), the compensation expression is evaluated.

Rule ATOMIC of λ_{FAIL} is replaced by the pair of rules PURE-ATOMIC and PURE-ATOMIC-ABORT, which describe the behavior of a transaction that is not contained within a workflow. PURE-ATOMIC describes the successful completion of a transaction, while PURE-ATOMIC-ABORT describes the case where the transaction is aborted. (The auxiliary relation $\twoheadrightarrow$ used here is defined by the rules for λ_{FAIL}.)

Compensation monad We now describe the *compensation monad* that can be used to realize workflows with compensating actions. For simplicity, we describe an implementation of the compensating monad that focuses only on logical failures and compensations. We assume there are no duplicate requests or process failures. We can realize idempotent workflows with compensating actions by *composing* this monad and the idempotence monad [13].

The compensation monad, shown in Fig. 9, is a combination of the exception monad and the continuation passing style monad. Transactions return a value of the form *Value*(v) (on successful completion) or a special value *Abort* to indicate that the transaction was aborted. Workflows are represented as a function in continuation passing style. The helper function *compensateWith* associates a transaction a with a compensating action to construct a primitive workflow. *compensateWith* constructs a function in continuation passing style. This function first evaluates $a()$. If a aborts, the whole transaction aborts and returns *Abort*. Otherwise, the continuation is evaluated using the value returned by the transaction. If the continuation itself aborts (because one of the following transactions aborts), we evaluate the compensating action and return the value *Abort*. The monad's *return* simply lifts a value (with no compensation) into a workflow. The monadic *bind* is standard for continuation passing style computations. The function *run* shows how to execute a workflow by passing it an empty continuation.

4.2 Asynchronous evaluation

Workflows are commonly used to perform computations involving several transactions. Consequently, workflows are often long running with highly variable latencies. Large latencies accentuate the

(a) Syntax and Evaluation Context

$$x \in \langle \mathsf{Identifier} \rangle; \qquad v \in \langle \mathsf{Val} \rangle \ ::= \ x \mid \lambda x.e$$
$$e \in \langle \mathsf{Exp} \rangle ::= x \mid \lambda x.e \mid e\ e \mid \texttt{atomic}\ v_t\ v_a\ v_c \mid \texttt{idworkflow}\ v_i\ v_w \mid \texttt{abort} \mid \texttt{update}\ v\ v \mid \texttt{lookup}\ v \mid e \oslash e_c$$
$$\Sigma_A = \langle \mathsf{Exp} \rangle \times \langle \mathsf{Exp} \rangle + \langle \mathsf{Exp} \rangle \times \langle \mathsf{Exp} \rangle$$
$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E \oslash e$$

(b) Evaluation Rules (in addition to A \ { ATOMIC })

[IDWF-BEGIN]
$$\frac{v \triangleright E[\texttt{idworkflow}\ id\ w] \in \alpha \qquad \nexists e'.id \blacktriangleright e' \in \alpha}{\langle p, \mu, \alpha, I, O\rangle \overset{\epsilon}{\Rightarrow} \langle p, \mu, \alpha \uplus \{id \blacktriangleright (w() \oslash 0)\}, I, O\rangle}$$

[IDWF-END]
$$\frac{id \blacktriangleright u \in \alpha \qquad u \in \langle \mathsf{Val} \rangle}{\langle p, \mu, \alpha \uplus \{v \triangleright E[\texttt{idworkflow}\ id\ w]\}, I, O\rangle \overset{\epsilon}{\Rightarrow} \langle p, \mu, \alpha \uplus \{v \triangleright E[u]\}, I, O\rangle}$$

[NORMAL]
$$\frac{\langle \mu, e\rangle \overset{\omega}{\rightsquigarrow} \langle \mu', e'\rangle}{\langle p, \mu, \alpha \uplus \{v \blacktriangleright e\}, I, O\rangle \Rightarrow \langle p, \mu', \alpha \uplus \{v \blacktriangleright e'\}, I, O\rangle}$$

[ATOMIC]
$$\frac{\langle \mu[tn], e_a()\rangle \overset{\omega}{\twoheadrightarrow}{}^{*} \langle t, v\rangle, v \in \langle \mathsf{Val} \rangle}{\langle \mu, E[\texttt{atomic}\ tn\ e_a\ e_c] \oslash e\rangle \overset{\omega}{\rightsquigarrow} \langle \mu[tn \mapsto t], E[v] \oslash ((e_c\ v); e)\rangle}$$

[ATOMIC-ABORT]
$$\frac{\langle \mu[tn], e_a()\rangle \overset{\omega}{\twoheadrightarrow}{}^{*} \langle t, E[\texttt{abort}]\rangle}{\langle \mu, E[\texttt{atomic}\ tn\ e_a\ e_c] \oslash e\rangle \overset{\epsilon}{\rightsquigarrow} \langle \mu, E[\texttt{abort}] \oslash e\rangle}$$

[IDWF-ABORT]
$$\frac{}{\langle \mu, (E[\texttt{abort}]) \oslash e_c\rangle \rightsquigarrow \langle \mu, e_c\rangle}$$

[CONTEXT]
$$\frac{e \to e'}{\langle \mu, E[e] \oslash e_c\rangle \to \langle \mu, E[e'] \oslash e_c\rangle}$$

[PURE-ATOMIC]
$$\frac{\langle \mu[tn], e_a()\rangle \overset{\omega}{\twoheadrightarrow}{}^{*} \langle t, v\rangle, v \in \langle \mathsf{Val} \rangle}{\langle \mu, \texttt{atomic}\ tn\ e_a\ e_c\rangle \overset{\omega}{\rightsquigarrow} \langle \mu[tn \mapsto t], v\rangle}$$

[PURE-ATOMIC-ABORT]
$$\frac{\langle \mu[tn], e_a()\rangle \overset{\omega}{\twoheadrightarrow}{}^{*} \langle t, E[\texttt{abort}]\rangle}{\langle \mu, \texttt{atomic}\ tn\ e_a\ e_c\rangle \overset{\epsilon}{\rightsquigarrow} \langle \mu, 0\rangle}$$

Figure 8: Syntax and Semantics of λ_{IDWF}.

```
let compensateWith a comp =
  fun f → match a() with
    | Abort → Abort
    | Value(b) → match (f b) with
        | Abort → let _ = comp b in Abort
        | Value(c) → Value(c)

let bind (v, f) = fun g → v (fun a → f a g)
let return a = compensateWith (fun () → a) (fun v → ())
let run a = a (fun x → Value(x))
```

Figure 9: The compensation monad

problem of duplicate requests because clients cannot easily distinguish between a long running workflow and one that has failed to generate a response. While the idempotence monad guarantees correctness in such cases, idempotence does come at a performance cost (due to log lookups). A design pattern commonly used to reduce the number of duplicate requests is for the system to take over the task of retrying (a part of) the request on behalf of the client *asynchronously*, and sending an intermediate response to the client, typically with the transaction identifier. The client can use transaction identifier to periodically poll for the status of the request.

The idempotence monad can be extended to support asynchronous evaluation as follows. At a programmer defined point in the evaluation of the workflow, we create a *checkpoint*. A checkpoint is essentially a closure representing the rest of the workflow, along with relevant state variables. The checkpoint is then persisted, typically in a distributed queue (essentially a worklist). Once the checkpoint has been created, an intermediate response is sent to the client. A set of special agents periodically query the queue, retrieve checkpoints, and continue evaluation, deleting the checkpoint only when the workflow has been fully evaluated.

Supporting asynchronous evaluation requires some additional support from the platform and a minor change to the monad's *bind* function. We assume that the platform provides a *channel* that can only be accessed by the idempotence monad (hence not exposed in λ_{FAIL}). Messages can be sent to this channel using the function *send* and received using the function *recv*. We assume the channel supports the following handshake protocol for messages in order to guarantee at-least-once processing of messages. In this protocol, *recv* does not delete messages from the channel. Instead, an agent that receives a message must acknowledge the message (using *ack*) before the message is permanently deleted. If an agent crashes after receiving a message and before acknowledging it, the message reappears in the channel and may be processed by other agents. (Windows Azure provides such a channel implementation, which our implementation uses.)

The changes to the idempotence monad are illustrated in Figure 10. Instead of invoking the remainder of the workflow, the modified *bind* (Fig. 10) creates a closure for the rest of the workflow and persists the closure in a special queue (*"worklist"*) (using the function *send*). Special agent processes (*agent*) retrieve the checkpoints (using *recv*) and continue evaluating the rest of workflow. If an agent manages to evaluate the workflow without failing, it deletes the workflow from the queue (using *ack*). In our implementation, the choice of using asynchronous evaluation (and the particular step at which to create a checkpoint) is left to the programmer. We evaluate the benefits of these optimizations in Section 5.

```
let agent () =
  while (true)
    let (msgid, msg) = recv "worklist"in
    let (f, params) = msg in
    let result = f params in
    let _ = ack "worklist" (msgid,msg) in result

let bind (idf, f) =
  fun (guid, tc) →
    let (ntc, v) = idf (guid, tc)
    send "worklist" (f a (guid, ntc + 1))
```

Figure 10: The idempotence monad with asynchronous evaluation

```
let saveSurvey response = idworkflow {
  do! addAtomic "responses" response.Id response
  return! atomic{
    let! summary = readAtomic "summaries" response.SurveyName
    do summary.Merge(response)
    return! writeAtomic "summaries" response.SurveyName summary }}
```

Figure 11: The save survey operation expressed as in idempotent workflow.

5. Evaluation

We have implemented the idempotence workflow monad and its variants in C# and F#, targeting Windows Azure, a platform for hosting distributed applications. Azure provides a key-value store with explicit support for data partitioning, where partitions are units of serializability. In this section, we focus on evaluating the expressiveness and performance overheads of the idempotence monad.

Sample applications We found several real world applications whose core logic can be declaratively expressed as workflows. We briefly describe an expense management application [4] and a survey application [3]. Other applications we have implemented include applications for online auctions [2], blogging and banking.

Survey application. The Surveys application enables users to design a survey, publish the survey, and collect the results of surveys. The most performance critical scenario in this application is when a user submits a survey. This operation involves two steps, recording the response and updating the survey summary with the response. For scalability, survey responses and summaries are stored in different data partitions. Figure 11 shows an implementation of this operation using workflows. The syntax here is from the actual F# implementation and differs from λ_{FAIL} in non-essential ways. The workflow is composed of two transactions, a transaction that saves the response and a transaction that updates the summary. In this implementation, survey responses do not reflect in the summary immediately. In general, *deferred writes* are often acceptable as long as the writes will eventually appear to occur exactly once, a guarantee workflows provide. The use of the idempotence monad guarantees that even if multiple requests are received with the same survey, the workflows appears to execute just once.

Auction application. The auction application allows users to bid for items on auction and track the status of their bids. Sellers can register items for auction with a time limit and a minimum bid price. Auction sites are often high volume sites, and both latency and scalability is important. The most critical operation in this application is the operation that processes new bids. This operation can be expressed as a workflow composed of several transaction.

	Hand-coded	**Molecules**
Banking	94	7296
Blogging	243	6484
Auction	107	7256
Surveys	530	9004
Expense	378	6504

Figure 12: Average message size (in bytes) of the hand-coded and workflow implementations.

The first transaction checks if the bid is valid and then checks if the bid beats the current maximum bid, in which case the bid is recorded in a bids table. The second transaction marks the current winner and loosing bidders in a separate table. The web front-end is programmed to poll this table periodically for the latest bid status. Subsequent transactions update other tables such as an aggregate table that maintains the most frequently bid items, more frequent bidders etc. With the idempotent workflow monad, it is easy to guarantee that each bid is processed exactly once even if clients retry with little change in complexity.

Overheads. There are two sources of overheads associated with idempotent workflows compared to hand-coded implementations. First, asynchronous evaluation requires serialization and deserialization closures, which can be expensive. Compiler generated closures tend to capture a lot more state than hand-coded implementations. Idempotent workflows also add unnecessary logging to transactions that are already idempotent.

Fig. 12 shows the average size of messages sent in hand-coded and workflow monad based implementations. As expected, the workflow based implementation generates significantly larger messages. However, experiments on Windows Azure show that the size of the message does not significantly influence the read/write latency or throughput of channels [1], as confirmed by our experiments below.

Next, we evaluate the overall performance overheads of using the idempotence monad relative to hand-coded implementations. For each benchmark application, we evaluate three versions - the original hand-coded version, and two monad based versions. The first version (*synchronous*) evaluates the workflow synchronously in the context of the current agent and delegates the responsibility of retrying operations to the client. The second version uses asynchronous evaluation.

We hosted each of the applications on Windows Azure and ensured that each variant was assigned the same hardware resources i.e. 5 virtual machines, each with a 4-core processor and 7 GB RAM. We assigned two virtual machines each for the front end agents that service browser requests and agents that interact with the storage system and one virtual machine for a background agent uses for asynchronous/check-pointed evaluation. In the synchronous evaluation variant, we assigned an additional virtual machine to the storage interaction agent. For each benchmark, we created a workload that simulates users exercising performance critical scenarios in the application. For example, in the surveys application, the workload consists of a mix of surveys response requests and survey analysis requests. In each case, the workload is biased towards write requests (e.g. 75% responses and 25% analysis requests in the survey application) to ensure that molecular transactions are on the critical path. We ran the workload for 3 minutes (with a 30 seconds warm-up period) and measured throughput (number of requests serviced per second) and latency (average response time) for varying number of simultaneous users.

Fig. 13 shows the measured throughput and latency for the surveys application. In the baseline version, both throughput and

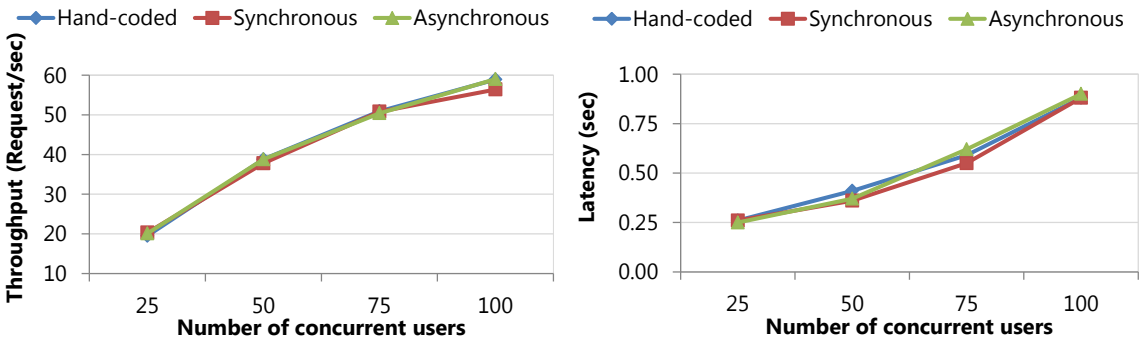

Figure 13: Average throughput and latency for various versions of the survey application.

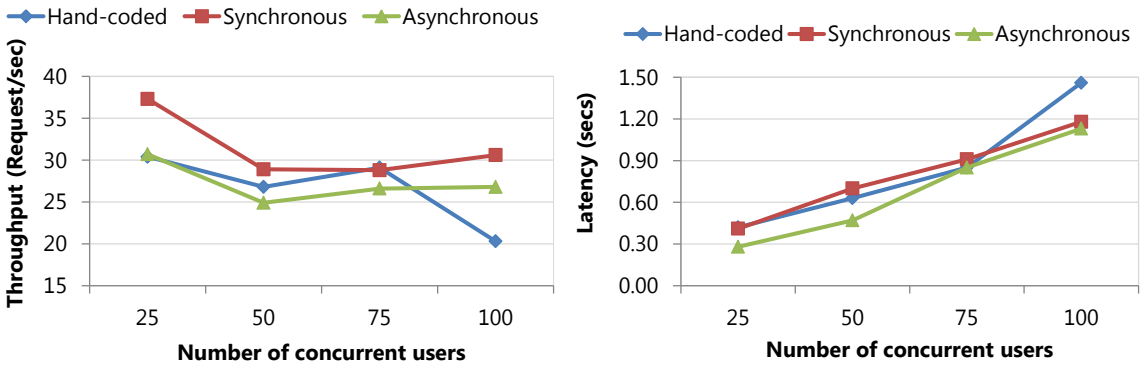

Figure 14: Average throughput and latency for various versions of the auction application.

latency increase almost linearly with the number of users. All monad based implementations closely follow this trend, suggesting very little additional overhead due to our implementation. The throughput (60 operations/second) and latencies are along expected lines. The expected throughput of Azure tables is 500 reads/writes per second, whereas our "operation" is a workflow consisting of two transactions.

The auction application (Fig. 14) has slightly different performance characteristics. The throughput of this application is lower than the surveys application and it decreases with load. This is expected since the workflow is significantly longer. Synchronous evaluation achieves better throughput than the hand-coded implemented (which is asynchronous) at almost the same latency. As expected, asynchronous evaluation improves latency, especially at low loads, but at the cost of reduced throughput (up to 25% lower). At high loads, all monad based implementations perform better than the hand-coded implementation. This is due to a few low level optimizations (such as batching of requests) performed by our implementation. We leave a more detailed performance analysis and optimization of the idempotence monad for future work.

6. Related Work

Our work builds on previous literature in the topics of building reliable distributed systems and transactional workflow. Idempotence has been widely and informally recognized as an important property of distributed systems. Our key contributions include: (a) A formal specification of the desired correctness properties, both safety (idempotence) as well as progress (failure freedom). (b) An automatic technique for guaranteeing failfree idempotence, presented as a monad. This implementation technique is decentralized and does not require distributed coordination. (c) Language constructs for idempotent workflow.

The need to move beyond atomic transactions to sequential compositions of atomic transactions (i.e., workflows) motivated the early work on Sagas and long running transactions [10, 20]. These constructs are weaker than distributed transactions and are generally used to orchestrate processes that run for extended periods. Kamath et al [12] discusses several issues relating to the implementation of transaction managers in Sagas and workflows. A distinguishing aspect of our work is that it exploits the fact that a service is required to be observationally idempotent (from its clients' perspective) to simplify the internal implementation of the workflow. In particular, this lets us avoid the need for distributed coordination with workflow managers.

Modern web applications often exploit horizontal partitioning for scalability, which involves storing data in a number of different databases or non-relational data-stores. This leads to a need for workflow-style computations even within a single intra-enterprise application. Since scalability is key (and the motivation for data partitioning), conventional workflow engines are rarely considered for use in this context. Instead, the programmer usually realizes the workflow manually, exposing themselves to all the subtleties of realizing such workflow correctly. Pritchett [16] describes programming methodologies for use in these scenarios. Our goal is to provide lightweight language mechanisms that can be used to realize idempotent workflows correctly in such scenarios.

Helland [11] explains in detail why idempotence is an essential property for reliable systems.

Frolund *et al.* [9] define a correctness criteria known as *X-Ability* for replicated services. A history is said to be *x-able* if it is equivalent to some history where every request is processed exactly once. Much like failfree idempotence, X-Ability is both a safety and liveness criteria. Our notion of failfree idempotence is generalizes X-Ability beyond replication requests to workflows. There are also significant differences in our implementation techniques.

Our work is also related to open nested multi-level transactions [15, 19]. These two constructs share the use of compensating actions, but are semantically different. Open nested transactions provide a way for dealing with conflicts at a higher level of abstraction, which often leads to increased concurrency.

Our basic setting is similar to Argus [14]. However, the construct that Argus provides programmers to deal with process failures is a conventional transaction. As with sagas, we show that many applications can be expressed using workflows (as we cover in Section 2) with compensating actions to compensate for the lack of isolation. The transactor programming model [7] also provide primitives for dealing with process failures in a distributed system. However, there is no shared state in the transactor model. The primitives provided by the transactor model (stabilize, checkpoint, and rollback) are different from the primitives we study.

Bruni et al [5] formalize compensating actions in an abstract language. Their formalism, however, does not explicitly model state. Their paper, in fact, suggests study of compensation in the context of imperative features (state, variables, control-flow constructs) as future work. Our work provides a compensation-based transactional construct as a library in a real language (F#) for a real system (Azure), in addition to the theoretical treatment in the setting of lambda-calculus with mutable shared state.

Luis et. al [6] propose an abstract model of compensating actions (based on process calculus) for reasoning about correctness for workflows that use compensating actions.

Faulty lambda calculus [18] is a programming language and a type system for fault tolerance. However, λ_{FAIL} is addresses process failures, while faulty lambda calculus addresses transient data corruption errors.

Acknowledgments

We would like to acknowledge Siddharth Agarwal, Nirmesh Malviya, Khilan Gudka, and Dheeraj Singh for their contributions.

References

[1] Azurescope: Benchmarking and Guidance for Windows Azure. `http://azurescope.cloudapp.net/BenchmarkTestCases/#4f2bdbcc-7c23-4c06-9c00-f2cc12d2d2a7`, June 2011.

[2] Bid Now Sample. `http://bidnow.codeplex.com`, June 2011.

[3] The Tailspin Scenario. `http://msdn.microsoft.com/en-us/library/ff966486.aspx`, June 2011.

[4] Windows Azure Patterns and Practices. `http://wag.codeplex.com/`, 2011.

[5] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *Proceedings of POPL*, pages 209–220, 2005.

[6] Luis Caires, Carla Ferreira, and Hugo Vieira. A Process Calculus Analysis of Compensations. In *Trustworthy Global Computing*, volume 5474 of *Lecture Notes in Computer Science*, pages 87–103. 2009.

[7] John Field and Carlos A. Varela. Transactors: A Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments. In *Proceedings of POPL*, pages 195–208, 2005.

[8] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[9] Svend Frolund and Rachid Guerraoui. X-Ability: A Theory of Replication. *Distributed Computing*, 14, 2000.

[10] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. of ICMD*, pages 249–259, 1987.

[11] Pat Helland. Idempotence is not a medical condition. *ACM Queue*, 10(4):30–46, 2012.

[12] Mohan Kamath and Krithi Ramamritham. Correctness Issues in Workflow Management. *Distributed Systems Engineering*, 3(4):213, 1996.

[13] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *In Proc. of POPL*, pages 333–343, 1995.

[14] Barbara Liskov. Distributed programming in argus. *Communications of ACM*, 31:300–312, March 1988.

[15] J. Eliot B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing, 1981.

[16] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[17] Philip Wadler and Peter Thiemann. The Marriage of Effects and Monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.

[18] David Walker, Lester Mackey, Jay Ligatti, George A. Reis, and David I. August. Static Typing for a Faulty Lambda Calculus. In *In ACM International Conference on Functional Programming*, 2006.

[19] Gerhard Weikum and Hans-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553, 1992.

[20] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. 2001.

A. An Abstract Definition of Idempotence

We now present a more abstract definition of idempotence in terms of histories.

Basics. The users of the system issue requests to the system. Let I denote a set of input messages (or *requests*). The system responds to requests with a *response*. Let O denote a set of output values. An output message or response is a pair $(i, o) \in I \times O$ indicating that the output value o is produced in response to input request i.

A *history* π is a sequence $e_1 e_2 \cdots e_k$ of requests and responses (i.e., each e_i can be either a request or a response). We will restrict our attention to histories that satisfy the simple property that every response corresponds to an earlier request in the history. We define a *specification* Φ to be a set of histories. In the sequel, we use a set of histories to specify desired safety properties of the system.

In the sequel, let $q \in I$ range over requests and let $r \in I \times O$ range over responses. Let α, β, and γ range over sequences of requests and responses.

DEFINITION A.1. *A specification Φ is said to be* asynchronous *if it satisfies the following properties.*

1. $\alpha\beta q\gamma \in \Phi \Rightarrow \alpha q\beta\gamma \in \Phi$.
2. $\alpha r\beta\gamma \in \Phi \Rightarrow \alpha\beta r\gamma \in \Phi$.

The above conditions are a natural restriction on specifications because of messaging delays that cannot be controlled. The above property is also related to the notion of linearizability. Given any sequential specification Φ_s, "linearizable Φ_s" can be defined as the smallest asynchronous specification that contains Φ_s. Every linearizable specification satifies the asynchronous property defined above, but not all asynchronous specifications are linearizable.

Idempotence. Two requests q_1 and q_2 (in the same history) are said to be *duplicates* if $q_1 = q_2$. Two responses (q_1, r_1) and (q_2, r_2) (in the same history) are said to be *duplicates* if $q_1 = q_2$.

DEFINITION A.2. *A specification Φ is said to* idempotent *iff:*

1. *Duplicate requests have no effect: $\alpha q \beta q \gamma \in \Phi$ iff $\alpha q \beta \gamma \in \Phi$.*
2. *Duplicates responses have the same value:*

$$\alpha(q, o_1)\beta(q, o_2)\gamma \in \Phi \Rightarrow o_1 = o_2.$$

3. *Duplicate responses are allowed: $\alpha r \beta r \gamma \in \Phi$ iff $\alpha r \beta \gamma \in \Phi$.*

(The above definition is intended for asynchronous specifications. Hence, the asynchrony conditions have been used to simplify the definition.)

Idempotence Closure. We define a history to be *repetitionless* if it contains no duplicate requests or duplicate responses. We define a specification Φ to be *repetitionless* if all histories in Φ are repetitionless.

Given a repetitionless specification Φ, we define its idempotence closure $idem(\Phi)$ to be the smallest specification that contains Φ and is idempotent.

We now summarize our goal: given a program p that satisfies a repetitionless specification Φ in the absence of process failures and duplicate requests, construct a program p' that satisfies $idem(\Phi)$ even in the presence of process failures and duplicate requests.

Extension The above definitions can be generalized to permit requests to be of the form (k, v), where k is a unique-identifier (key) for an input request, and allowing responses to be of the form (k, o), where k is the unique-identifier of the input request for which the response is produced. Note that the above definition does not capture the progress conditions of *failure-freedom*.

B. The Semantics of $\lambda_{\mathbf{IDFF}}$

We adapt the earlier operational semantics of λ_{FAIL} as shown in Fig. 15 to define the semantics of λ_{IDFF}. We extend the definition Σ_A, the set of agents: previously, an agent was of the form $v \triangleright e$. Now, an agent may now also be of the form $v \blacktriangleright e$, indicating a `idff` evaluation of expression e with identifier v.

Syntax Extension: $e \in \langle \mathsf{Exp} \rangle ::= \cdots \mid \texttt{idff}\ v_1\ v_2$

Semantic Domain Changes: $\Sigma_A = \langle \mathsf{Exp} \rangle \times \langle \mathsf{Exp} \rangle + \langle \mathsf{Exp} \rangle \times \langle \mathsf{Exp} \rangle$

Additional Evaluation Rules:

[IDFF-NEW]
$$\frac{v \triangleright E[\texttt{idff}\ id\ e] \in \alpha \qquad \nexists e'.id \blacktriangleright e' \in \alpha}{\langle p, \mu, \alpha, I, O\rangle \stackrel{\epsilon}{\Rightarrow} \langle p, \mu, \alpha \uplus \{id \blacktriangleright e()\}, I, O\rangle}$$

[IDFF-USE]
$$\frac{id \blacktriangleright w \in \alpha \qquad w \in \langle \mathsf{Val} \rangle}{\langle p, \mu, \alpha \uplus \{v \triangleright E[\texttt{idff}\ id\ e]\}, I, O\rangle \stackrel{\epsilon}{\Rightarrow} \langle p, \mu, \alpha \uplus \{v \triangleright E[w]\}, I, O\rangle}$$

[NORMAL]
$$\frac{\langle \mu, e\rangle \stackrel{\ell}{\rightsquigarrow} \langle \mu', e'\rangle}{\langle p, \mu, \alpha \uplus \{v \blacktriangleright e\}, I, O\rangle \Rightarrow \langle p, \mu', \alpha \uplus \{v \blacktriangleright e'\}, I, O\rangle}$$

Figure 15: The language λ_{IDFF}, defined via extensions to language λ_{FAIL}. Rules IDFF-NEW and IDFF-USE must be duplicated for $v \blacktriangleright E[\texttt{idff}\ id\ e]$ as well.

Rule IDFF-NEW handles the evaluation of the construct `idff` $v\ e$, when no preceding `idff` computation with the same identifier v has been initiated. This has the effect of creating a new agent $v \blacktriangleright e$. Rule IDFF-USE allows the computation `idff` $v\ e$ to proceed once the created agent completes evaluation (as indicated by the presence of an agent of the form $v \blacktriangleright w$, where w is a value). The same rule also applies to "duplicate" evaluations with the same id.