CPL

A Core Language for Cloud Computing

Bračevac, Oliver; Erdweg, Sebastian; Salvaneschi, Guido; Mezini, Mira

DOI
10.1145/2889443.2889452

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings of the 15th International Conference on Modularity, Modularity 2016

Citation (APA)
Bračevac, O., Erdweg, S., Salvaneschi, G., & Mezini, M. (2016). CPL: A Core Language for Cloud Computing. In Proceedings of the 15th International Conference on Modularity, Modularity 2016 (pp. 94-105). New York, NY: Association for Computing Machinery (ACM). https://doi.org/10.1145/2889443.2889452

CPL: A Core Language for Cloud Computing

Oliver Bračevac[1] Sebastian Erdweg[1] Guido Salvaneschi[1] Mira Mezini[1,2]
[1]TU Darmstadt, Germany [2]Lancaster University, UK

Abstract

Running distributed applications in the cloud involves deployment. That is, distribution and configuration of application services and middleware infrastructure. The considerable complexity of these tasks resulted in the emergence of declarative JSON-based domain-specific *deployment languages* to develop *deployment programs*. However, existing deployment programs unsafely compose artifacts written in different languages, leading to bugs that are hard to detect before run time. Furthermore, deployment languages do not provide extension points for custom implementations of existing cloud services such as application-specific load balancing policies.

To address these shortcomings, we propose CPL (*Cloud Platform Language*), a statically-typed core language for programming both distributed applications as well as their deployment on a cloud platform. In CPL, application services and deployment programs interact through statically typed, extensible interfaces, and an application can trigger further deployment at run time. We provide a formal semantics of CPL and demonstrate that it enables type-safe, composable and extensible libraries of *service combinators*, such as load balancing and fault tolerance.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems; D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Cloud deployment; cloud computing; computation patterns; join calculus

1. Introduction

Cloud computing [30] has emerged as the reference infrastructure for concurrent distributed services with high availability, resilience and quick response times, providing access to on-demand and location-transparent computing resources. Companies develop and run distributed applications on specific cloud platforms, e.g., Amazon AWS[1] or Google Cloud Platform.[2] Services are bought as needed from the cloud provider in order to adapt to customer demand,

An important and challenging task in the development process of cloud applications is *deployment*. Especially, deployment involves the distribution, configuration and composition of (1) virtual machines that implement the application and its services, and of (2) virtual machines that provide middleware infrastructure such as load balancing, key-value stores, and MapReduce. Deploying a cloud application can go wrong and cause the application to malfunction. Possible causes are software bugs in the application itself, but also wrong configurations, such as missing library dependencies or inappropriate permissions for a shell script. Fixing mistakes *after* deployment causes high costs and loss of reputation. For example, in 2012, Knight Capital lost over $440 Million over the course of 30 minutes due to a bug in its deployed trading software,[3] causing the disappearance of the company from the market.

Considering that cloud applications can have deployment sizes in the hundreds or thousands of virtual machines, manual configuration is error-prone and does not scale. Cloud platforms address this issue with domain-specific languages (DSLs) such as Amazon CloudFormation or Google Cloud Deployment Manager. The purpose of these DSLs is to write reusable *deployment programs*, which instruct the cloud platform to perform deployment steps automatically. A typical deployment program specifies the required virtual machines for the application infrastructure, how these virtual machines are connected with each other, and how the application infrastructure connects to the pre-existing or newly created middleware infrastructure of the cloud platform.

[1] `https://aws.amazon.com`

[2] `https://cloud.google.com`

[3] `http://www.bloomberg.com/bw/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes`.

However, the modularity of current cloud deployment DSLs is insufficient (detailed discussion in Section 2):

Unsafe Composition: Application services and deployment programs are written in different languages. Deployment DSLs configure application services by lexically expanding configuration parameters into application source code before its execution. This approach is similar to a lexical macro system and makes deployment programs unsafe because of unintentional code injection and lexical incompatibilities.

No Extensibility: Middleware cloud services (e.g., elastic load balancing, which may dynamically allocate new virtual machines) are pre-defined in the cloud platform and only referenced by the deployment program through external interfaces. As such, there is no way to customize those services during deployment or extend them with additional features.

Stage Separation: Current deployment DSLs finish their execution before the application services are started. Therefore, it is impossible to change the deployment once the application stage is active. Thus, applications cannot self-adjust their own deployment, e.g., to react to time-varying customer demand.

We propose CPL (Cloud Platform Language), a statically-typed core language for programming cloud applications and deployments. CPL employs techniques from programming language design and type systems to overcome the issues outlined above. Most importantly, CPL unifies the programming of deployments and applications into a single language. This avoids unsafe composition because deployments and applications can exchange values directly via statically typed interfaces. For extensibility, CPL supports higher-order service combinators with statically typed interfaces using bounded polymorphism. Finally, CPL programs run at a single stage where an application service can trigger further deployment.

To demonstrate how CPL solves the problems of deployment languages, we implemented a number of case studies. First, we demonstrate type-safe composition through generically typed worker and thunk abstractions. Second, on top of the worker abstraction, we define composable and reusable *service combinators* in CPL, which add new features, such as elastic load balancing and fault tolerance. Finally, we demonstrate how to model MapReduce as a deployment program in CPL and apply our combinators, obtaining different MapReduce variants, which safely deploy at run time.

In summary, we make the following contributions:

- We analyze the problems with current cloud deployment DSLs.
- We define the formal syntax and semantics of CPL to model cloud platforms as distributed, asynchronous message passing systems. Our design is inspired by the Join Calculus [12].
- We define the type system of CPL as a variant of System F with bounded quantification [24].

```
1  { //...
2   "Parameters": {
3    "InstanceType": {
4     "Description": "WebServer EC2 instance type",
5     "Type": "String",
6     "Default": "t2.small",
7     "AllowedValues": [ "t2.micro", "t2.small ]",
8     "ConstraintDescription": "a valid EC2 instance type."
9    } //...
10  },
11  "Resources": {
12    "WebServer": {
13      "Type": "AWS::EC2::Instance",
14      "Properties": {
15        "InstanceType": { "Ref" : "InstanceType" }  ,
16        "UserData": { "Fn::Base64" : { "Fn::Join" : ["", [
17              "#!/bin/bash -xe\n",
18              "yum update -y aws-cfn-bootstrap\n",
19
20              "/opt/aws/bin/cfn-init -v ",
21              "   --stack ", { "Ref" : "AWS::StackName" },
22              "   --resource WebServer ",
23              "   --configsets wordpress_install ",
24              "   --region ", { "Ref" : "AWS::Region" }, "\n"
25        ]]}}, //...
26      },
27    }
28  },
29  "Outputs": {
30    "WebsiteURL": {
31      "Value":
32        { "Fn::Join" :
33          ["", ["http://", { "Fn::GetAtt" :
34            [ "WebServer", "PublicDnsName" ]}, "/wordpress" ]]},
35      "Description" : "WordPress Website"
36    }
37  }
38 }
```

Figure 1. A deployment program in CloudFormation (details omitted, full version: `https://s3.eu-central-1.amazonaws.com/cloudformation-templates-eu-central-1/WordPress_Multi_AZ.template`.

- We formalize CPL in PLT Redex [9] and we provide a concurrent implementation in Scala.
- We evaluated CPL with case studies, including a library of typed service combinators that model elastic load balancing and fault tolerance mechanisms. Also, we apply the combinators to a MapReduce deployment specification.

The source code of the PLT Redex and Scala implementations and of all case studies is available online: `https://github.com/seba--/djc-lang`.

2. Motivation

In this section, we analyze the issues that programmers encounter with current configuration and deployment languages on cloud platforms by a concrete example.

2.1 State of the Art

Figure 1 shows an excerpt of a deployment program in CloudFormation, a JSON-based DSL for Amazon AWS. The example is from the CloudFormation documentation. We summarize the main characteristics of the deployment language below.

- Input parameters capture varying details of a configuration (Lines 2-10). For example, the program receives the

virtual machine instance type that should host the web server for a user blog (Line 3). This enables reuse of the program with different parameters.

- CloudFormation programs specify *named resources* to be created in the deployment (Lines 11-28), e.g., deployed virtual machines, database instances, load balancers and even other programs as modules. The program in Figure 1 allocates a "WebServer" resource (Line 12), which is a virtual machine instance. The type of the virtual machine references a parameter (Line 15), that the program declared earlier on (Line 3). Resources can refer to each other, for example, configuration parameters of a web server may refer to tables in a database resource.
- Certain configuration phases require to execute application code inside virtual machine instances after the deployment stage. Application code is often directly specified in resource bodies (Lines 17-24). In the example, a bash script defines the list of software packages to install on the new machine instance (in our case a WordPress[4] blog). In principle, arbitrary programs in any language can be specified.
- Deployment programs specify output parameters (Lines 29-37), which may depend on input parameters and resources. Output parameters are returned to the caller after executing the deployment program. In this example, it is a URL pointing to the new WordPress blog.
- The deployment program is interpreted at run time by the cloud platform which performs the deployment steps according to the specification.

2.2 Problems with Deployment Programs

In the following we discuss the issues with the CloudFormation example described above.

Internal Safety Type safety for deployment programs is limited. Developers define "types" for resources of the cloud platform, e.g., `AWS::EC2::Instance` (Line 13) represents an Amazon EC2 instance. However, the typing system of current cloud deployment languages is primitive and only relies on the JSON types.

Cross-language Safety Even more problematic are issues caused by cross-language interaction between the deployment language and the language(s) of the deployed application services. For example, the `AWS::Region` variable is passed from the JSON specification to the bash script (Line 24). However, the sharing mechanism is just syntactic replacement of the current value of `AWS::Region` inside the script. Neither are there type-safety checks nor syntactic checks before the script is executed. More generally, there is no guarantee that the data types of the deployment language are compatible with the types of the application language nor that the resulting script is syntactically correct. This problem makes cloud applications susceptible to hygiene-related bugs and injection attacks [4].

Low Abstraction Level Deployment languages typically are Turing-complete but the abstractions are low-level and not deployment-specific. For example, (1) deployment programs receive parameters and return values similar to procedures and (2) deployment programs can be instantiated from inside other deployment programs, which resembles modules. Since deployment is a complex engineering task, advanced language features are desirable to facilitate programming in the large, e.g., higher-order combinators, rich data types and strong interfaces.

Two-phase Staging Deployment programs in current DSLs execute before the actual application services, that is, information flows from the deployment language to the deployed application services, but not the other way around. As a result, an application service cannot reconfigure a deployment based on the run time state. Recent scenarios in reactive and big data computations demonstrate that this is a desirable feature [10].

Lack of Extensibility Resources and service references in deployment programs refer to pre-defined abstractions of the cloud platform, which have rigid interfaces. Cloud platforms determine the semantics of the services. Programmers cannot implement their own variants of services that plug into the deployment language with the same interfaces as the native services.

Informal Specification The behavior of JSON deployment scripts is only informally defined. The issue is exacerbated by the mix of different languages. As a result, it is hard to reason about properties of systems implemented using deployment programs.

The issues above demand for a radical change in the way programmers deploy cloud applications and in the way application and deployment configuration code relate to each other.

3. The Cloud Platform Language

A solution to the problems identified in the previous section requires an holistic approach where cloud abstractions are explicitly represented in the language. Programmers should be able to modularly specify application behavior as well as reconfiguration procedures. Run time failures should be prevented at compilation time through type checking.

These requirements motivated the design of CPL. In this section, we present its syntax and the operational semantics.

3.1 Language Features in a Nutshell

Simple Meta-Theory: CPL should serve as the basis for investigating high-level language features and type systems designed for cloud computations. To this end, it is designed as a core language with a simple meta-theory. Es-

[4] `http://wordpress.org`

tablished language features and modeling techniques, such as lexical scoping and a small-step operational semantics, form the basis of CPL.

Concurrency: CPL targets distributed concurrent computations. To this end, it allows the definition of independent computation units, which we call *servers*.

Asynchronous Communication: Servers can receive parameterized *service requests* from other servers. To realistically model low-level communication within a cloud, the language only provides asynchronous end-to-end communication, where the success of a service request is not guaranteed. Other forms of communication, such as synchronous, multicast, or error-checking communication, can be defined on top of the asynchronous communication.

Local Synchronization: Many useful concurrent and asynchronous applications require synchronization. We adopt *join patterns* from the Join Calculus [12]. Join patterns are *declarative* synchronization primitives for machine-local synchronization.

First-Class Server Images: Cloud platforms employ virtualization to spawn, suspend and duplicate virtual machines. That is, virtual machines are data that can be stored and send as payload in messages. This is the core idea behind CPL's design and enables programs to change their deployment at run time. Thus, CPL features servers as values, called *first-class servers*. Active *server instances* consist of an address, which points to a *server image* (or *snapshot*). The server image embodies the current run time state of a server and a description of the server's functionality, which we call *server template*. At run time, a server instance may be overwritten by a new server image, thus changing the behavior for subsequent service requests to that instance.

Transparent Placement: Cloud platforms can reify new machines physically (on a new network node) or virtually (on an existing network node). Since this difference does not influence the semantics of a program but only its non-functional properties (such as performance), our semantics is transparent with respect to placement of servers. Thus, actual languages based on our core language can freely employ user-defined placement definitions and automatic placement strategies. Also, we do not require that CPL-based languages map servers to virtual machines, which may be inefficient for short-lived servers. Servers may as well represent local computations executing on a virtual machine.

3.2 Core Syntax

Figure 2 displays the core syntax of CPL. An expression e is either a value or one of the following syntactic forms:[5]

- A *variable* x is from the countable set $\mathcal{N}$. Variables identify services and parameters of their requests.
- A *server template* $(\mathbf{srv}\ \overline{r})$ is a first-class value that describes the behavior of a server as a sequence of reaction rules $\overline{r}$. A reaction rule takes the form $\overline{p} \triangleright e$, where $\overline{p}$ is a sequence of joined service patterns and e is the body. A *service pattern* $x_0\langle\overline{x}\rangle$ in $\overline{p}$ declares a service named x_0 with parameters $\overline{x}$ and a rule can only fire if all service patterns are matched simultaneously. The same service pattern can occur in multiple rules of a server.
- A *server spawn* $(\mathbf{spwn}\ e)$ creates a new running *server instance* at a freshly allocated *server address* i from a given *server image* $(\mathbf{srv}\ \overline{r}, \overline{m})$ represented by e. A *server image* is a description of a server behavior plus a server state – a *buffer* of pending messages. A real-world equivalent of server images are e.g., virtual machine snapshots. A special case of a server image is the value $\mathbf{0}$, which describes an inactive or shut down server.
- A fully qualified *service reference* $e\sharp x$, where e denotes a server address and x is the name of a service provided by the server instance at e. Service references to server instances are themselves values.
- A *self-reference* $\mathbf{this}$ refers to the address of the lexically enclosing server template, which, e.g., allows one service to call upon other services of the same server instance.
- An asynchronous *service request* $e_0\langle\overline{e}\rangle$, where e_0 represents a service reference and $\overline{e}$ the arguments of the requested service.
- A *parallel expression* $(\mathbf{par}\ \overline{e})$ of service requests $\overline{e}$ to be executed independently. The empty parallel expression $(\mathbf{par}\ \varepsilon)$ acts as a noop expression, unit value, or null process and is a value.
- A *snapshot* $\mathbf{snap}\ e$ yields an image of the server instance which resides at the address denoted by e.
- A *replacement* $\mathbf{repl}\ e_1\ e_2$ of the server instance at address e_1 with the server image e_2.

Notation: In examples, p `&` p denotes pairs of join patterns and $e \parallel e$ denotes pairs of parallel expressions. We sometimes omit empty buffers when spawning servers, i.e., we write $\mathbf{spwn}\ (\mathbf{srv}\ \overline{r})$ for $\mathbf{spwn}\ (\mathbf{srv}\ \overline{r}, \varepsilon)$. To improve readability in larger examples, we use curly braces to indicate the lexical scope of syntactic forms. We write service names and meta-level definitions in typewriter font, e.g., $\mathbf{this}\sharp$`foo` and `MyServer =` $\mathbf{srv}$ { }. We write bound variables in italic font, e.g., $\mathbf{srv}$ { `left`$\langle x\rangle$ `&` `right`$\langle y\rangle \triangleright$ `pair`$\langle x,\ y\rangle$ }.

[5] We write $\overline{a}$ to denote the finite sequence $a_1 \ldots a_n$ and we write ε to denote the empty sequence.

$$
\begin{array}{lll}
e \;::=\; v \mid x \mid \mathbf{this} \mid \mathbf{srv}\ \overline{r} \mid \mathbf{spwn}\ e \mid e\sharp x \mid e\langle \overline{e}\rangle \mid \mathbf{par}\ \overline{e} \\
\qquad \mid \mathbf{snap}\ e \mid \mathbf{repl}\ e\ e & \text{(Expressions)} & i \in \mathbb{N} \qquad \text{(Server Addresses)} \\
 & & r \;::=\; \overline{p} \triangleright e \qquad \text{(Reaction Rules)} \\
v \;::=\; \mathbf{srv}\ \overline{r} \mid i \mid i\sharp x \mid \mathbf{par}\ \varepsilon \mid (\mathbf{srv}\ \overline{r}, \overline{m}) \mid \mathbf{0} & \text{(Values)} & p \;::=\; x\langle \overline{x}\rangle \qquad \text{(Join Patterns)} \\
 & & m \;::=\; x\langle \overline{v}\rangle \qquad \text{(Message Values)} \\
E \;::=\; [\cdot] \mid \mathbf{spwn}\ E \mid E\sharp x \mid E\langle \overline{e}\rangle \mid e\langle \overline{e}\ E\ \overline{e}\rangle & \text{(Evaluation Contexts)} & \\
\qquad \mid \mathbf{par}\ \overline{e}\ E\ \overline{e} \mid \mathbf{snap}\ E \mid \mathbf{repl}\ E\ e \mid \mathbf{repl}\ e\ E & & \mu \;::=\; \varnothing \mid \mu; i \mapsto (\mathbf{srv}\ \overline{r}, \overline{m}) \qquad \text{(Routing Tables)} \\
 & & \qquad \mid \mu; i \mapsto \mathbf{0} \\
x, y, z \in \mathcal{N} & \text{(Variable Names)} &
\end{array}
$$

Figure 2. Expression Syntax of CPL.

$$
\frac{e \mid \mu \;\longrightarrow\; e' \mid \mu'}{E[e] \mid \mu \;\longrightarrow\; E[e'] \mid \mu'} \quad (\textsc{Cong})
$$

$$
\frac{}{\mathbf{par}\ \overline{e_1}\ (\mathbf{par}\ \overline{e_2})\ \overline{e_3} \mid \mu \;\longrightarrow\; \mathbf{par}\ \overline{e_1}\ \overline{e_2}\ \overline{e_3} \mid \mu} \quad (\textsc{Par})
$$

$$
\frac{\mu(i) = (s, \overline{m})}{i\sharp x\langle \overline{v}\rangle \mid \mu \;\longrightarrow\; \mathbf{par}\ \varepsilon \mid \mu; i \mapsto (s, \overline{m} \cdot x\langle \overline{v}\rangle)} \quad (\textsc{Rcv})
$$

$$
\frac{\mu(i) = (s, \overline{m}) \qquad s = \mathbf{srv}\ \overline{r}_1\ (\overline{p} \triangleright e_b)\ \overline{r_2} \qquad \mathrm{match}(\overline{p}, \overline{m}) \Downarrow (\overline{m}', \sigma) \qquad \sigma_b = \sigma \cup \{\mathbf{this} := i\}}{\mathbf{par}\ e \mid \mu \;\longrightarrow\; \mathbf{par}\ e\ \sigma_b(e_b) \mid \mu; i \mapsto (s, \overline{m}')} \quad (\textsc{React})
$$

$$
\frac{i \notin \mathrm{dom}(\mu) \qquad (s = \mathbf{0} \vee s = (\mathbf{srv}\ \overline{r}, \overline{m}))}{\mathbf{spwn}\ s \mid \mu \;\longrightarrow\; i \mid \mu; i \mapsto s} \quad (\textsc{Spwn})
$$

$$
\frac{\mu(i) = s \qquad (s = \mathbf{0} \vee s = (\mathbf{srv}\ \overline{r}, \overline{m}))}{\mathbf{snap}\ i \mid \mu \;\longrightarrow\; s \mid \mu} \quad (\textsc{Snap})
$$

$$
\frac{i \in \mathrm{dom}(\mu) \qquad (s = \mathbf{0} \vee s = (\mathbf{srv}\ \overline{r}, \overline{m}))}{\mathbf{repl}\ i\ s \mid \mu \;\longrightarrow\; \mathbf{par}\ \varepsilon \mid \mu; i \mapsto s} \quad (\textsc{Repl})
$$

Matching Rules:

$$
\frac{}{\mathrm{match}(\varepsilon, \overline{m}) \Downarrow (\overline{m}, \emptyset)} \quad (\textsc{Match}_0)
$$

$$
\frac{\overline{m} = \overline{m}_1\ (x\langle v_1 \ldots v_k\rangle)\ \overline{m}_2 \qquad \sigma = \{x_i := v_i \mid 1 \le i \le k\} \qquad \mathrm{match}(\overline{p}, \overline{m}_1\ \overline{m}_2) \Downarrow (\overline{m}_r, \sigma_r) \qquad \mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma_r) = \emptyset}{\mathrm{match}(x\langle x_1 \ldots x_k\rangle\ \overline{p}, \overline{m}) \Downarrow (\overline{m}_r, \sigma \cup \sigma_r)} \quad (\textsc{Match}_1)
$$

Figure 3. Small-step Operational Semantics of CPL.

Example. For illustration, consider the following server template `Fact` for computing factorials, which defines three rules with 5 services.[6]

```
1  Fact = srv {
2    main⟨n, k⟩ ▷ //initialization
3      this♯fac⟨n⟩ ‖ this♯acc⟨1⟩ ‖ this♯out⟨k⟩
4
5    fac⟨n⟩ & acc⟨a⟩ ▷ //recursive fac computation
6      if (n ≤ 1)
7      then this♯res⟨a⟩
8      else (this♯fac⟨n - 1⟩ ‖ this♯acc⟨a * n⟩)
9
10   res⟨n⟩ & out⟨k⟩ ▷ k⟨n⟩ //send result to k
11 }
```

The first rule defines a service `main` with two arguments, an integer n and a continuation k. The continuation is necessary because service requests are asynchronous and thus, the factorial server must notify the caller when the computation finishes. Upon receiving a `main` request, the server sends itself three requests: `fac` represents the outstanding factorial computation, `acc` is used as an accumulator for the ongoing computation, and `out` stores the continuation provided by the caller.

The second rule of `Fact` implements the factorial function and synchronously matches and consumes requests `fac` and `acc` using join patterns. Upon termination, the second rule sends a request `res` to the running server instance, otherwise it decreases the argument of `fac` and updates the accumulator. Finally, the third rule of `Fact` retrieves the user-provided continuation k from the request `out` and the result `res`. The rule expects the continuation to be a service reference and sends a request to it with the final result as argument.

To compute a factorial, we create a server instance from the template `Fact` and request service `main`:

$(\mathbf{spwn}\ \texttt{Fact})\sharp\texttt{main}\langle 5,\ k\rangle.$

3.3 Operational Semantics

We define the semantics of CPL as a small-step structural operational semantics using reduction contexts E (Figure 2) in the style of Felleisen and Hieb [8].

Figure 3 shows the reduction rules for CPL expressions. Reduction steps are atomic and take the form $e \mid \mu \longrightarrow e' \mid \mu'$. A pair $e \mid \mu$ represents a distributed cloud application, where expression e describes its current behavior and μ describes its current *distributed* state. We intend e as a description of the software components and resources that execute and reside at the cloud provider and do not model client devices. We call the component μ a *routing table*, which is a finite map. Intuitively, μ records which addresses a cloud provider assigns to the server instances that the cloud application creates during its execution.[7] We abstract over technical details, such as the underlying network.

[6] For the sake of presentation, we use ordinary notation for numbers, arithmetics and conditionals, all of which is church-encodable on top of CPL (cf. Section 3.5).

[7] This bears similarity to lambda calculi enriched with references and a store [31].

The first reduction rule (CONG) defines the congruence rules of the language and is standard. The second rule (PAR) is technical. It flattens nested parallel expressions in order to have a simpler representation of parallel computations. The third rule (RCV) lets a server instance receive an asynchronous service request, where the request is added to the instance's buffer for later processing. Our semantics abstracts over the technicalities of network communication. That is, we consider requests $i\sharp x\langle\overline{v}\rangle$ that occur in a CPL expression to be in transit, until a corresponding (RCV) step consumes them. The fourth rule (REACT) fires reaction rules of a server. It selects a running server instance $(s, \overline{m})$, selects a reaction rule $(\overline{p} \triangleright e_b)$ from it, and tries to match its join patterns $\overline{p}$ against the pending service requests in the buffer $\overline{m}$. A successful match consumes the service requests, instantiates the body e_b of the selected reaction rule and executes it independently in parallel.

Finally, let us consider the rules for **spwn**, **snap** and **repl**, which manage server instances and images. Reduction rule (SPWN) creates a new server instance from a server image, where a fresh unique address is assigned to the server instance. This is the only rule that allocates new addresses in μ. One can think of this rule as a request to the cloud provider to create a new virtual machine and return its IP address. Importantly, the address that **spwn** yields is only visible to the caller. The address can only be accessed by another expression if it shares a common lexical scope with the caller. Thus, lexical scope restricts the visibility of addresses. This also means that the map μ is not a shared memory, but a combined, flat view of disjoint distributed information.[8]

Reduction rule (SNAP) yields a copy of the server image at address i, provided the address is in use. Intuitively, it represents the invocation of a cloud management API to create a virtual machine snapshot. Reduction rule (REPL) replaces the server image at address i with another server image.

We define **spwn**, **snap** and **repl** as atomic operations. At the implementation level, each operation may involve multiple communication steps with the cloud provider, taking noticeable time to complete and thus block execution for too long, especially when the operation translates to booting a new OS-level virtual machine. On the other hand, as we motivated at the beginning of this section, servers may not necessarily map to virtual machines, but in-memory computations. In this case, we expect our three atomic operations to be reasonably fast. Also, we do not impose any synchronization mechanism on a server addresses, which may result in data races if multiple management operations access it in parallel. Instead, programmers have to write their own synchronization mechanisms on top of CPL if required.

Our semantics is nondeterminstic along 3 dimensions:

- If multiple server instances can fire a rule, (REACT) selects one of them nondeterminstically. This models concurrent execution of servers that can react to incoming service requests independently.
- If multiple rules of a server instance can fire, (REACT) selects one of them nondeterminstically. This is of lesser importance and languages building on ours may fix a specific order for firing rules (e.g., in the order of definition).
- If multiple service request values can satisfy a join pattern, (MATCH_1) selects one of them nondeterminstically. This models asynchronous communication in distributed systems, i.e., the order in which a server serves requests is independent of the order in which services are requested. More concrete languages based on CPL may employ stricter ordering (e.g., to preserve the order of requests that originate from a single server).

3.4 Placement of Servers

We intentionally designed the semantics of CPL with transparency of server *placement* in mind. That is, a single abstraction in the language, the server instance, models all computations, irrespective of whether the instance runs on its own physical machine or as a virtual machine hosted remotely – indeed, placement transparency is a distinguishing feature of cloud applications.

However, despite the behavior of servers being invariant to placement, placement has a significant impact in real-world scenarios and influences communication and computation performance [2, 19]. The need to account for placement in an implementation is critical considering that – servers being the only supported abstraction – every single let binding and lambda abstraction desugars to a server spawn (cf. Section 3.5). In our concurrent Scala implementation, we support an extended syntax for server spawns that allows programmers to declare whether a server instance runs in a new thread or in the thread that executes the spawn. This provides a simple mechanism for manually implementing placement strategies.

A viable alternative to manual specification of placement are automatic placement strategies. Together with server migration, automatic placement strategies can adapt the server layout to changing conditions. Based on our language, a management system for a cloud infrastructure can formally reason about optimal placement strategies. In future work, we plan to implement these ideas in a distributed run-time system for CPL (cf. Section 4.4).

3.5 Derived Syntax and Base Operations

For notational convenience in examples, we will assume a fully-fledged, practical programming language, knowing that in principle we can encode all required features in core CPL. Our core language is sufficiently expressive to encode typical language constructs on top of message passing, such as let bindings, λ abstractions, numbers, base operations, thunks

[8] This approach is comparable to sets of definitions in the chemical soup of the Join Calculus [12].

(first-class value that represents a packaged, delayed computation) and algebraic data types. As the basis we adopted the CPS function encoding from the Join Calculus [12], which is defined in our technical report [3]. For example, we desugar ($\rightsquigarrow$) standard call-by-value **let** expressions to server spawns and service requests:
$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \ \rightsquigarrow\ (\mathbf{spwn}\ (\mathbf{srv}\ \texttt{let}\langle x\rangle \triangleright e_2))\sharp\texttt{let}\langle e_1\rangle$.
The desugared **let** spawns a new server instance with a `let` service that triggers the body e_2. This service is requested with the bound expression e_1 as an argument. If e_1 reduces to a value v, then the whole expression evaluates to $e_2\,\{x := v\}$ eventually. Otherwise, the expression is stuck.

Another variant of **let** is **letk**, which is convenience syntax to specify anonymous continuations for service requests:
$\mathbf{letk}\ x = e_1\langle\overline{e}\rangle\ \mathbf{in}\ e_2 \rightsquigarrow e_1\langle\overline{e}, (\mathbf{spwn}\ (\mathbf{srv}\ \texttt{k}\langle x\rangle \triangleright e_2))\sharp\texttt{k}\rangle$.
Intuitively, a **letk** expression triggers an asynchronous service e_1, passing the arguments $\overline{e}$. The body e_2 is the continuation after the e_1 request yields a value, which is bound to x in e_2.

Sometimes it is useful to delay the evaluation of an expression, e.g., to store and transfer expressions between servers or to implement lazy evaluation. We use the convenience syntax $\mathbf{thunk}\ e$ to denote delayed expressions:
$\mathbf{thunk}\ \ e \rightsquigarrow \mathbf{srv}\ \texttt{force}\langle k\rangle \triangleright k\langle e\rangle$. The encoding wraps the expression e in a server template, which is a value. In order to evaluate e, one spawns $\mathbf{thunk}\ e$ and requests the `force` service.

3.6 Type System

We designed and formalized a type system for CPL in the style of System F with subtyping and bounded quantification [24]. The type system ensures that all service requests in a well-typed program refer to valid service declarations with the correct number of arguments and the right argument types. As such, the type system statically ensures well-typed compositions of servers and services, of which the JSON-based deployment languages we analyzed in Section 2 are incapable.

Structural subtyping enables us to define public server interfaces, where the actual server implementation defines private services, for example, to manage internal state and swap implementations at run time.

A detailed description of the type system rules and soundness proofs are in the accompanying technical report [3]. We sketch a few examples to prepare the reader for the subsequent discussions. Typed CPL adds annotations to each service at the beginning of server templates:

$$S := \mathbf{srv}\ \texttt{a}{:}\langle\mathsf{Int}\rangle,\ \texttt{b}{:}\langle\langle\mathsf{Bool}\rangle\rangle \\ (\texttt{a}\langle x\rangle \triangleright \texttt{foo}\langle\rangle)\ (\texttt{a}\langle x\rangle\ \texttt{\&}\ \texttt{b}\langle y\rangle \triangleright \texttt{bar}\langle\rangle),$$

where service $S\sharp\texttt{a}$ is of the service type $\langle\mathsf{Int}\rangle$, accepting an integer value and $S\sharp\texttt{b}$ is of service type $\langle\langle\mathsf{Bool}\rangle\rangle$ accepting a boolean-typed continuation. The entire type of the server template S is the structural type $\mathbf{srv}\ \texttt{a}{:}\langle\mathsf{Int}\rangle\ \texttt{b}{:}\langle\mathsf{Bool}\rangle$. A spawn $\mathbf{spwn}\ (S, \overline{m})$ yields a server address of type $\mathbf{inst}\ \mathbf{srv}\ \texttt{a}{:}\langle\mathsf{Int}\rangle\ \texttt{b}{:}\langle\mathsf{Bool}\rangle$. Our subtyping relation permits S in contexts that require the less informative interfaces $\mathbf{srv}\ \texttt{a}{:}\langle\mathsf{Int}\rangle$, $\mathbf{srv}\ \texttt{b}{:}\langle\mathsf{Bool}\rangle$ or $\mathbf{srv}\ \varepsilon$.

For notational convenience, we define function types as

$$T_1, \ldots T_n \to T := \langle T_1, \ldots, T_n, \langle T\rangle\rangle$$

following the function encoding in Section 3.5.

4. CPL at Work

We present two case studies to demonstrate the adequacy of CPL for solving the deployment issues identified in Section 2. The case studies will also be subsequently used to answer research questions about CPL's features.

Firstly, we developed a number of reusable *server combinators*, expressing deployment patterns found in cloud computing. Our examples focus on load balancing and fault tolerance, demonstrating that programmers can define their own cloud services as strongly-typed, composable modules and address nonfunctional requirements with CPL. Secondly, we use our language to model MapReduce [18] deployments for distributed batch computations. Finally, we apply our server combinators to MapReduce, effortlessly obtaining a type-safe composition of services.

4.1 Server Combinators

In Section 2, we identified extensibility issues with deployment languages, which prevents programmers from integrating their own service implementations. We show how to implement custom service functionality with *server combinators* in a type-safe and composable way. Our combinators are similar in spirit to higher-order functions in functional programming.

As the basis for our combinators, we introduce **workers**, i.e., servers providing computational resources. A worker accepts work packages as thunks. Concretely, a worker models a managed virtual machine in a cloud and thunks model application services.

Following our derived syntax for thunks (Section 3.5), given an expression e of type α, the type of $\mathbf{thunk}\ e$ is:

$$\mathsf{TThunk}[\alpha] := \mathbf{srv}\ \texttt{force}{:}\langle\alpha\rangle.$$

Service `force` accepts a continuation and calls it with the result of evaluating e. A worker accepts a thunk and executes it. At the type level, workers are values of a polymorphic type

$$\mathsf{TWorker}[\alpha] := \mathbf{srv}\ \texttt{init}{:}\langle\rangle,\ \texttt{work}{:}\mathsf{TThunk}[\alpha] \to \alpha.$$

That is, to execute a thunk on a worker, clients request the `work` service which maps the thunk to a result value. In addition, we allow workers to provide initialization logic via a service `init`. Clients of a worker should request `init` before they issue `work` requests. Figure 4 defines a factory for creating basic workers, which have no initialization logic and execute thunks in their own instance scope. In the following, we define server combinators that enrich workers with more advanced features.

```
1  MkWorker[α] = srv {
2   make: () → TWorker[α]
3   make⟨k⟩ ▷
4    let worker = spwn srv {
5     init: ⟨⟩, work: TThunk[α] → α
6     init⟨⟩ ▷ par ε //stub, do nothing
7     work⟨thnk, k⟩ ▷ (spwn thnk)♯force⟨k⟩
8    } in k⟨worker⟩
9  }
```

Figure 4. Basic worker factory.

To model **locality** – a worker uses its own computational resources to execute thunks – the spawn of a thunk should in fact *not* yield a new remote server instance. As discussed in Section 3.4, to keep the core language minimal the operational semantics does not distinguish whether a server is local or remote to another server. However, in our concurrent implementation of CPL, we allow users to annotate spawns as being remote or local, which enables us to model worker-local execution of thunks.

The combinators follow a common design principle. *(i)* The combinator is a factory for server templates, which is a server instance with a single `make` service. The service accepts one or more server templates which implement the TWorker interface, among possibly other arguments. *(ii)* Our combinators produce *proxy* workers. That is, the resulting workers implement the worker interface but forward requests of the *work* service to an internal instance of the argument worker.

4.1.1 Load Balancing

A common feature of cloud computing is on-demand scalability of services by dynamically acquiring server instances and distributing load among them. CPL supports the encoding of on-demand scalability in form of a server combinator, that distributes load over multiple workers dynamically, given a user-defined decision algorithm.

Dynamically distributing load requires a means to approximate worker utilization. Our first combinator `MkLoadAware` enriches workers with the ability to answer `getLoad` requests, which sends the current number of pending requests of the `work` service, our measure for utilization. Therefore, the corresponding type[9] for load aware workers is

$$\mathsf{TLAWorker}[\alpha] := \mathsf{TWorker}[\alpha] \cup \mathbf{srv}\ \texttt{getLoad:}\langle\langle \mathsf{Int}\rangle\rangle.$$

The `make` service of the combinator accepts a server template *worker* implementing the TWorker interface and returns its enhanced version (bound to *lWorker*) back on the given continuation k. Lines 10-13 implement the core idea of forwarding and counting the pending requests. Continuation passing style enables us to intercept and hook on to the

[9] The union ∪ on server types is for notational convenience at the meta level and *not* part of the type language.

```
1  MkLoadAware[α, ω <: TWorker[α]] = srv {
2   make: ω → TLAWorker[α]
3   make⟨worker, k⟩ ▷
4    let lWorker = srv {
5     instnc: ⟨inst ω⟩, getLoad: () → Int, load: ⟨Int⟩
6     work: TThunk[α] → α, init: ⟨⟩
7     //... initialization logic omitted
8
9     //forwarding logic for work
10    work⟨thnk, k⟩ & instnc⟨w⟩ & load⟨n⟩ ▷
11     this♯load⟨n+1⟩ ‖ this♯instnc⟨w⟩
12     ‖ letk res = w♯work⟨thnk⟩
13        in (k⟨res⟩ ‖ this♯done⟨⟩)
14
15    //callback logic for fullfilled requests
16    done⟨⟩ & load⟨n⟩ ▷ this♯load⟨n-1⟩
17
18    getLoad⟨k⟩ & load⟨n⟩ ▷ k⟨n⟩ ‖ this♯load⟨n⟩
19   } in k⟨lWorker⟩
20 }
```

Figure 5. Combinator for producing load-aware workers.

responses of *worker* after finishing work requests, which we express in Line 13 by the **letk** construct.

By building upon load-aware workers, we can define a polymorphic combinator `MkBalanced` that transparently introduces load balancing over a list of load-aware workers. The combinator is flexible in that it abstracts over the scheduling algorithm, which is an impure polymorphic function of type

$$\mathsf{Choose}[\omega] := \mathsf{List}[\mathbf{inst}\ \omega] \to \mathsf{Pair}[\mathbf{inst}\ \omega, \mathsf{List}[\mathbf{inst}\ \omega]].$$

Given a (church-encoded) list of possible worker instances, such a function returns a (church-encoded) pair consisting of the chosen worker and an updated list of workers, allowing for dynamic adjustment of the available worker pool (*elastic load balancing*).

Figure 6 shows the full definition of the `MkBalanced` combinator. Similarly to Figure 5, the combinator is a factory which produces a decorated worker. The only difference being that now there is a list of possible workers to forward requests to. Choosing a worker is just a matter of querying the scheduling algorithm *choose* (Lines 15-16). Note that this combinator is only applicable to server templates implementing the TLAWorker[α] interface (Line 1), since *choose* should be able to base its decision on the current load of the workers.

In summary, mapping a list of workers with `MkLoadAware` and passing the result to `MkBalanced` yields a composite, load-balancing worker. It is thus easy to define hierarchies of load balancers programmatically by repeated use of the two combinators. Continuation passing style and the type system enable flexible, type-safe compositions of workers.

4.1.2 Failure Recovery

Cloud platforms monitor virtual machine instances to ensure their continual availability. We model failure recovery for

```
1  MkBalanced[α, ω <: TLAWorker[α]] = srv {
2   make: (List[ω], Choose[ω]) → TWorker[α]
3   make⟨workers, choose, k⟩ ▷
4    let lbWorker = srv {
5     insts: ⟨List[inst ω]⟩,
6     work: TThunk[α] → α, init: ⟨⟩
7
8     init⟨⟩ ▷ //spawn and init all child workers
9      letk spawned = mapk⟨workers, λw:ω. spwn w⟩
10      in (this♯insts⟨spawned⟩
11          ‖foreach⟨spawned, λinst:inst ω. inst♯init⟨⟩⟩)
12
13    //forward to the next child worker
14    work⟨thnk, k⟩ & insts⟨l⟩ ▷
15     letk (w, l′) = choose⟨l⟩
16      in (w♯work⟨thnk, k⟩ ‖ this♯insts⟨l′⟩)
17   } in k⟨lbWorker⟩
18 }
```

Figure 6. Combinator for producing load-balanced workers.

crash/omission, permanent, fail-silent failures [27], where a failure makes a virtual machine unresponsive and is recovered by a restart.

Following the same design principles of the previous section, we can define a failure recovery combinator `MkRecover`, that produces fault-tolerant workers. We omit its definition and refer to our technical report [3].

Self-recovering workers follow a basic protocol. Each time a `work` request is processed, we store the given thunk and continuation in a list until the underlying worker confirms the request's completion. If the wait time exceeds a timeout, we replace the worker with a fresh new instance and replay all pending requests. Crucial to this combinator is the **repl** syntactic form, which swaps the running server instance at the worker's address: **repl** w $(worker, \varepsilon)$. Resetting the worker's state amounts to setting the empty buffer ε in the server image value we pass to **repl**.

4.2 MapReduce

In this section, we illustrate how to implement the MapReduce [7] programming model with typed combinators in CPL, taking fault tolerance and load balancing into account. MapReduce facilitates parallel data processing – cloud platforms are a desirable deployment target. The main point we want to make with this example is that CPL programs do not exhibit the unsafe composition, non-extensibility and staging problems we found in Section 2. Our design is inspired by Lämmel's formal presentation in Haskell [18].

Figure 7 shows the main combinator for creating a MapReduce deployment, which is a first-class server. Following Lämmel's presentation, the combinator is generic in the key and value types. κ_i denotes type parameters for keys and ν_i denotes type parameters for values.

```
1  MapReduce[κ1,ν1,κ2,ν2,ν3] = spwn srv {
2   make: (TMap[κ1,ν1,κ2,ν2],
3          TReduce[κ2,ν2,ν3],
4          TPartition[κ2],
5          ∀α.() → TWorker[α],
6          Int) → TMR[κ1,ν1,κ2,ν3]
7
8   make⟨Map, Reduce, Partition, R, mkWorker, k⟩ ▷
9    let sv = srv {
10    app⟨data, k0⟩ ▷ let
11     mworker =
12       mapValues(data, λv. mkWorker[List[Pair[κ1, ν2]]])
13     rworker =
14       mkMap(map(range(1, R), λi. (i, mkWorker[ν3])))
15     grouper =
16       MkGrouper⟨Partition, R, Reduce
17                 size(mworker), rworker, k0⟩
18    in foreach⟨data, λkey, val. {
19     let thnk = thunk Map⟨key, val⟩
20      in get(mworker, key)♯work⟨thnk, grouper♯group⟩}⟩
21   } in k⟨sv⟩
22 }
```

Figure 7. MapReduce factory.

The combinator takes as parameters (1) the Map function for decomposing input key-value pairs into a list of intermediate pairs, (2) the $Reduce$ function for transforming grouped intermediate values into a final result value, (3) the $Partition$ function which controls grouping and distribution among reducers, and (4) the number R of reducers to allocate (Line 8). Parameter $mkWorker$ is a polymorphic factory of type $\forall\alpha.() \rightarrow \mathsf{TWorker}[\alpha]$. It produces worker instances for both the map and reduce stage.

Invoking `make` creates a new server template that on invocation of its `app` service deploys and executes a distributed MapReduce computation. That is, it processes the given set of (church-encoded) key-value pairs $data$ and returns the result on continuation k_0 (Lines 9-10).

Firstly, workers for mapping and reducing are allocated and stored in the local map data structures $mworker$ and $rworker$, where we assume appropriate cps-encoded functions that create and transform maps and sequences (Lines 11-14). Each key in the input $data$ is assigned a new mapping worker and each partition from 1 to R is assigned a reducing worker. Additionally, a component for grouping and distribution among reducers ($grouper$) is allocated. We refer to our technical report [3] for the definition of the grouper.

Secondly, the `foreach` invocation (Lines 18-20) distributes key-value pairs in parallel among mapping workers. For each pair, the corresponding worker should invoke the Map function, which we express as a thunk (Line 19, cf. section 3.5). All resulting intermediate values are forwarded to the grouper's `group` service.

Thanks to our service combinators, we can easily address non-functional requirements and non-intrusively add new fea-

tures. The choice of the $mkWorker$ parameter determines which variants of MapReduce deployments we obtain: The default variant just employs workers without advanced features, i.e.,

```
let make = Λα.(spwn MkWorker[α])♯make
in MapReduce[κ1,ν1,κ2,ν2,ν3]♯make⟨f, r, p, R, make, k⟩
```

for appropriate choices of the other MapReduce parameters.

In order to obtain a variant, where worker nodes are elastically load-balanced, one replaces $make$ with $makeLB$ below, which composes the combinators from the previous section:

```
let choose = ...//load balancing algorithm
    makeLB = Λα.λk. {
     letk w = (spwn MkWorker[α])♯make⟨⟩
       lw = (spwn MkLoadAware[α, TWorker[α]])♯make⟨w⟩
      in (spwn MkBalanced[α, TLAWorker[α]])
              ♯make⟨mkList(lw), choose, k⟩
    }
in makeLB
```

A similar composition with the fault tolerance combinator yields fault tolerant MapReduce, where crashed mapper and reducer workers are automatically recovered.

4.3 Discussion

We discuss how CPL performed in the case studies answering the following research questions:

Q1 (Safety): *Does CPL improve safety of cloud deployments?*

Q2 (Extensibility): *Does CPL enable custom and extensible service implementations?*

Q3 (Dynamic self-adjustment): *Does CPL improve flexibility in dynamic reconfiguration of deployments ?*

Safety CPL is a strongly-typed language. As such, it provides internal safety (Section 2). The issue of cross-language safety (Section 2) does not occur in CPL programs, because configuration and deployment code are part of the same application. In addition, the interconnection of components is well-typed. For example, in the MapReduce case study, it is guaranteed that worker invocations cannot go wrong due to wrongly typed arguments. It is also guaranteed that workers yield values of the required types. As a result, all mapper and reducer workers are guaranteed to be compatible with the grouper component. In a traditional deployment program, interconnecting components amounts to referring to each others attributes, but due to the plain syntactic expansion, there is no guarantee of compatibility.

Extensibility The possibility to define combinators in CPL supports extensible, custom service implementations. At the type system level, bounded polymorphism and subtyping ensure that service implementations implement the required interfaces. The load balancing example enables nested load balancing trees, since the combinator implements the well-known Composite design pattern from object-oriented programming. At the operational level, continuation passing style enables flexible composition of components, e.g., for stacking multiple features.

Dynamic Self-Adjustment In the case studies, we encountered the need of dynamically adapting the deployment configuration of an application, which is also known as "elasticity". For example, the load balancer combinator can easily support dynamic growth or shrinkage of the list of available workers: New workers need to be dynamically deployed in new VMs (growth) and certain VMs must be halted and removed from the cloud configuration when the respective workers are not needed (shrinkage). Dynamic reconfiguration is not directly expressible in configuration languages, due to the two-phase staging. For example, configurations can refer to external elastic load balancer services provided by the cloud platform, but such services only provide a fixed set of balancing strategies, which may not suit the application. The load balancer service can be regarded as a black box, which happens to implement elasticity features. Also, a configuration language can request load balancing services only to the fixed set of machines which is specified in a configuration, but it is not possible if the number of machines is unknown before execution, as in the MapReduce case study. In contrast, CPL users can specify their own load balancing strategies and apply them programmatically.

4.4 Interfacing with Cloud Platforms

A practical implementation of CPL requires (1) a mapping of its concepts to real-world cloud platforms and (2) integrate existing cloud APIs and middleware services written in other languages. In the following, we sketch a viable solution; we leave a detailed implementation for future work.

For (1), CPL programs can be compiled to bytecode and be interpreted by a distributed run time hosted on multiple virtual machines.

Concerning (2), we envision our structural server types as the interface of CPL's run time with the external world, i.e., pre-existing cloud services and artifacts written in other languages. CPL developers must write wrapper libraries to implement typed language bindings. Indeed, CPL's first-class servers resemble (remote) objects, where services are their methods and requests are asynchronous method invocations (returning results on continuations). CPL implementations hence can learn from work on language bindings in existing object-oriented language run times, e.g., the Java ecosystem. To ensure type safety, dynamic type checking is necessary at the boundary between our run time and components written in dynamically or weakly typed languages.

Note that the representation of external services and artifacts as servers requires *immutable* addresses. That is, the run time should forbid **snap** and **repl** on such objects, because it is in general impossible to reify a state snapshot of the external world.

For the primitives **spwn**, **snap**, and **repl**, the run time must be able to orchestrate the virtualization facilities of the

cloud provider via APIs. Following our annotation-based approach to placement (Section 3.4), these primitives either map to local objects or to fresh virtual machines. Thus, invoking **spwn** v may create a new virtual machine hosting the CPL run time, which allocates and runs v. For local servers, v is executed by the run time that invoked **spwn**. One could extend the primitives to allow greater control of infrastructure-level concerns, such as machine configuration and geographic distribution. From these requirements and CPL's design targeting extensible services and distributed applications, it follows that CPL is cross-cutting the three abstraction layers in contemporary cloud platforms: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [30].

5. Related Work

Programming Models for Cloud Computing. The popularity of cloud computing infrastructures [30] has encouraged the investigation of programming models that can benefit from on-demand, scalable computational power and feature location transparency. Examples of these languages, often employed in the context of big data analysis, are Dryad [16], PigLatin [23] and FlumeJava [6]. These languages are motivated by refinements and generalizations of the original MapReduce [7] model.

Unlike CPL, these models specifically target only certain kinds of cloud computations, i.e., massive parallel computations and derivations thereof. They deliberately restrict the programming model to enable automated deployment, and do not address deployment programmability in the same language setting as CPL does. In this paper, we showed that the server abstraction of CPL can perfectly well model MapReduce computations in a highly parametric way, but it covers at the same time a more generic application programming model as well as deployment programmability. Especially, due to its join-based synchronization, CPL is well suited to serve as a core language for modeling cloud-managed stream processing.

Some researchers have investigated by means of formal methods specific computational models or specific aspects of cloud computing. The foundations in functional programming of MapReduce have been studied by Lämmel [18]. In CPL it is possible to encode higher-order functions and hence we can model MapReduce's functionality running on a cloud computing platform. Jarraya et al. [17] extend the Ambient calculus to account for firewall rules and permissions to verify security properties of cloud platforms. To the best of our knowledge, no attempts have been done in formalizing cloud infrastructures in their generality.

Formal Calculi for Concurrent and Distributed Services. Milner's CCS [20], the π calculus [21] and Hoare's CSP [15] have been studied as the foundation of parallel execution and process synchronization.

Fournet's and Gonthier's Join Calculus [12] introduced join patterns for expressing the interaction among a set of processes that communicate by asynchronous message passing over communication channels. The model of communication channels in this calculus more adequately reflects communication primitives in real world computing systems which allows for a simpler implementation. In contrast, the notion of channel in the previously mentioned process calculi would require expensive global consensus protocols in implementations.

The design of CPL borrows join patterns from the Join Calculus. Channels in the Join Calculus are similar to services in CPL, but the Join Calculus does not have first-class and higher-order servers with qualified names. Also, there is no support for deployment abstractions.

The Ambient calculus [5] has been developed by Cardelli and Gordon to model concurrent systems that include both mobile devices and mobile computation. Ambients are a notion of named, bounded places where computations occur and can be moved as a whole to other places. Nested ambients model administrative domains and capabilities control access to ambients. CPL, in contrast, is location-transparent, which is faithful to the abstraction of a singular entity offered by cloud applications.

Languages for Parallel Execution and Process Synchronization. Several languages have been successfully developed/extended to support features studied in formal calculi.

JoCaml is an ML-like implementation of Join Calculus which adopts state machines to efficiently support join patterns [11]. Polyphonic C# [1] extends C# with join-like concurrency abstractions for asynchronous programming that are compiler-checked and optimized. Scala Joins [14] uses Scala's extensible pattern matching to express joins. The Join Concurrency Library [26] is a more portable implementation of Polyphonic C# features by using C# 2.0 generics. JEScala [29] combines concurrency abstraction in the style of the Join Calculus with implicit invocation.

Funnel [22] uses the Join Calculus as its foundations and supports object-oriented programming with classes and inheritance. Finally, JErlang [25] extends the Erlang actor-based concurrency model. Channels are messages exchanged by actors, and received patterns are extended to express matching of multiple subsequent messages. Turon and Russo [28] propose an efficient, lock-free implementation of the join matching algorithm demonstrating that declarative specifications with joins can scale to complex coordination problems with good performance – even outperforming specialized algorithms. Fournet et al. [13] provide an implementation of the Ambient calculus. The implementation is obtained through a formally-proved translation to JoCaml.

CPL shares some features with these languages, basically those built on the Join Calculus. In principle, the discussion about the relation of CPL to Join Calculus applies to these languages as well, since the Join Calculus is their shared

foundation. Implementations of CPL can benefit from the techniques developed in this class of works, especially [26].

6. Conclusions and Future Work

We presented CPL, a statically typed core language for defining asynchronous cloud services and their deployment on cloud platforms. CPL improves over the state of the art for cloud deployment DSLs: It enables (1) statically safe service composition, (2) custom implementations of cloud services that are composable and extensible and (3) dynamic changes to a deployed application. In future work, we will implement and expand core CPL to a practical programming language for cloud applications and deployment.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work has been supported by the European Research Council, grant No. 321217.

References

[1] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM TOPLAS*, 26(5):769–804, Sept. 2004.

[2] N. Bobroff, A. Kochut, and K. A. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007.

[3] O. Bračevac, S. Erdweg, G. Salvaneschi, and M. Mezini. CPL: A Core Language for Cloud Computing. Technical report, Software Technology Group, Technische Universität Darmstadt, `http://arxiv.org/abs/1602.00981`, 2016.

[4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing Injection Attacks with Syntax Embeddings. GPCE '07, pages 3–12. ACM, 2007.

[5] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177 – 213, 2000.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. PLDI '10, pages 363–375, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] M. Felleisen and R. Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.

[9] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[10] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD '13*, pages 725–736. ACM, June 2013.

[11] F. L. Fessant and L. Maranget. Compiling Join-Patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205 – 224, 1998. HLCL'98.

[12] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96*, pages 372–385. ACM, 1996.

[13] C. Fournet, J.-J. Lévy, and A. Schmitt. An Asynchronous, Distributed Implementation of Mobile Ambients. TCS '00, pages 348–364. Springer-Verlag, 2000.

[14] P. Haller and T. Van Cutsem. Implementing Joins Using Extensible Pattern Matching. In *COORDINATION '08*, volume 5052 of *LNCS*, pages 135–152. Springer, 2008.

[15] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[16] M. Isard and Y. Yu. Distributed Data-parallel Computing Using a High-level Programming Language. SIGMOD '09, pages 987–994. ACM, 2009.

[17] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. Cloud calculus: Security verification in elastic cloud computing platform. In *CTS'12*, pages 447–454, May 2012.

[18] R. Lämmel. Google's MapReduce programming model — Revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[19] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *INFOCOM*, pages 1154–1162. IEEE, 2010.

[20] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[21] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1 – 40, 1992.

[22] M. Odersky. An Introduction to Functional Nets. In *Applied Semantics*, volume 2395 of *LNCS*, pages 333–377. Springer, 2002.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. SIGMOD '08, pages 1099–1110. ACM, 2008.

[24] B. C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[25] H. Plociniczak and S. Eisenbach. JErlang: Erlang with joins. In *COORDINATION '10*, volume 6116 of *LNCS*, pages 61–75. Springer, 2010.

[26] C. Russo. The joins concurrency library. In *PADL '07*, volume 4354 of *LNCS*, pages 260–274. Springer, 2007.

[27] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[28] A. J. Turon and C. V. Russo. Scalable Join Patterns. In *OOPSLA '11*, pages 575–594. ACM, 2011.

[29] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular Coordination with Declarative Events and Joins. MODULARITY '14, pages 205–216. ACM, 2014.

[30] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, Dec. 2008.

[31] A. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38 – 94, 1994. ISSN 0890-5401.