Type Safe Evolution of Live Systems

Miguel Domingues
NOVA-LINCS – Universidade Nova de Lisboa
miguel@domingues.pt

João Costa Seco
NOVA-LINCS – Universidade Nova de Lisboa
joao.seco@fct.unl.pt

ABSTRACT

This paper introduces a novel programming model for safe and incremental construction of live applications. We capture a verified style of agile development that spans over the whole development life cycle, from specification and prototyping to maintenance and evolution. This approach proposes a step forward with relation to the traditional *code-compile-deploy* cycle, allowing for both code and data updates to be safely applied during execution. We propose a language-based development and runtime system to evolve data-centric applications. Our approach is presented as a core typed imperative calculus with a reactive semantics. The associated type discipline ensures a correct interleaving of interaction and construction of systems. The soundness of our calculus is supported by standard progress, type preservation and convergence results.

1. INTRODUCTION

Development frameworks and agile methodologies are increasingly popular, especially in the domain of web and cloud applications. Such tools provide support for code refactoring, but do not avoid the risk of service disruption that a software update may cause. Updates are a natural part of the software system's life cycle, one that is particularly prone to errors, that may cause significant downtime and impact on end users [3]. Such effect is magnified by the gap between the code and the persistent state. The act of programming is mostly based on the developer's reasoning, whose results are only visible in a separate moment while debugging, testing and running the application. Moreover, updates in data-centric applications often require explicitly programmed scripts to handle the evolution of both code and persistent data, in an ad-hoc and synchronized way.

We present an effective and typeful incremental way of constructing applications without causing service disruption. We introduce a novel core programming model that provides immediate feedback on the effects of gradually introducing new programming elements, or redefining existing ones. Our model combines the static verification of each programming step with a scheduling discipline that ensures the absence of runtime errors and interferences between execution and development. Such goals are instantiated in a core programming language that allows for a flexible style of incremental [14] and live [4] programming, where typing ensures that the system is always sound. Live programming environments are set to reduce the feedback loop that exists between writing the code and perceiving its effects. We extend this goal to the complete life cycle of data-centric systems, where updates are defined using a uniform reconfiguration mechanism, that includes both code and data.

Reactive frameworks and software architectures (e.g. Meteor[1]) are increasingly important in the definition of collaborative applications, as well as in other computational models [2]. However, the way that changes in persistent data are transmitted to user interfaces, is usually implemented by ad-hoc and hand-crafted code. Using suitable language constructs, enable us to specify and reason about the evolution of applications. Our model is instantiated in a imperative and reactive programming language, capturing the essence of reactive frameworks, and imperative characteristics of typical data-centric applications, allowing to continuously evolve code [7, 11], and reconfigure the underlying data [3, 8].

To the best of our knowledge, this is the first work combining a type-safe imperative and reactive language with a statically verified and uniform evolution mechanism. Our key contributions are:

- a novel core programming model for data-centric applications, that supports type-safe dynamic evolution.
- a type system that statically ensures the safety of applications in the presence of reactive propagation and dynamic evolution.
- a data-flow operational semantics supporting the synchronized evolution of both code and data.
- type preservation and progress results that imply properties like convergence of change propagation.

In Section 2 we introduce our model with an example. Section 3 formally presents our core programming language, type system, operational semantics, and incremental construction mechanism. Section 4 states the soundness results of our language. Sections 5 and 6 provide a comparison with related work, and final remarks.

2. LIVE AND REACTIVE PROGRAMMING

We introduce three main ingredients to model our target scenario of data-centric applications: state variables, data transformation expressions, and actions. Bound *state variables* model the persistent data layer. Application logic layer is modeled by bound *data transformation expressions*, representing either a query over persistent data, or code that processes results to serve a view of the system. *Actions* are delayed computations modeling event handlers, and enclose (imperative) insert or update queries to the data layer.

We follow a layered model where we provide a remote interface (console) to manage the application code (c.f. [11]). Construc-

[1] www.meteor.com

$$\textbf{var}\ \text{messages}=[\left\{\begin{array}{ll} id & =0,\\ author & =\text{“Paul”},\\ message & =\text{“Hi there! ...”},\\ likes & =10 \end{array}\right\},\ldots]$$

$$\textbf{def}\ \text{size}=\textbf{foreach}(x\ \textbf{in}\ \text{messages}\ \textbf{with}\ y=0)\ y+1$$

$$\textbf{def}\ \text{post}=\lambda a.\lambda m.\textbf{action}\ \{\textbf{insert}\ \left\{\begin{array}{ll} id & =\text{size},\\ author & =a,\\ message & =m,\\ likes & =0 \end{array}\right\}\ \textbf{into}\ \text{messages}\}$$

$$\textbf{def}\ \text{like}=\lambda i.\textbf{action}\ \{\textbf{update}\ m\ \textbf{in}\ \text{messages}\ \textbf{with}\ \left\{\begin{array}{ll} id & =m.id,\\ author & =m.author,\\ message & =m.message,\\ likes & =m.likes+1 \end{array}\right\}\ \textbf{where}\ m.id=i\}$$

$$\textbf{var}\ \text{user}=\text{“Paul”}$$

$$\textbf{def}\ \text{wall}=\textbf{map}(r\ \textbf{in}\ \text{messages})\ \left\{\begin{array}{ll} m & =r,\\ incLikes & =\text{like}\ r.id \end{array}\right\}$$

$$\textbf{def}\ \text{index}=\textbf{map}(r\ \textbf{in}\ \text{wall})\ \{\ldots\textbf{do}\ r.incLikes\ldots\}\ldots\{\ldots\textbf{do}\ \text{post user}\ \#\texttt{message}\ldots\}$$

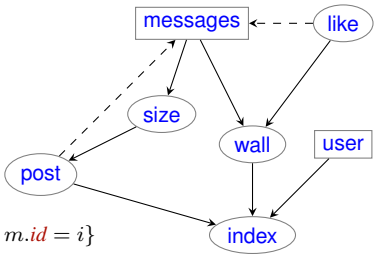

Figure 1: Bulletin Board.

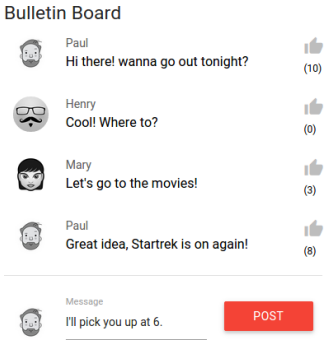

Figure 2: Bulletin Board UI.

tion operations are interpreted at the systems' interface level to (re)define parts of an application, and trigger data updates. Operations submitted to the system are statically checked and ill-typed operations are rejected. Moreover, we follow a data-driven operational semantics where changes in data are pro-actively pushed to interfaces. A working prototype of a live development environment based on this model can be found in [1], where it is possible to build and evolve applications, while at the same time interact with them.

We next illustrate the syntax and semantics of our language by means of a simple example, developed in two different stages.

2.1 Building the System

Consider the problem of developing a small bulletin board application, with a user interface (UI) similar to Figure 2. This application includes a list of messages stored persistently; a "thumbs-up" icon for each message, linked to actions that increment the counter of "likes" for that message; a form (text box and a button "Post") linked to an action that adds a new message. Figure 1 shows the construction operations that may be used to support such UI.

We first define state variable messages containing a collection ([...]) of sample messages, records ({...}). This defines the initial (persistent) state of the application, and is incrementally applied to the running system. Each record in collection messages contains a message identifier (*id*), the *author*, the *message*, and the number of *likes*. The scope of name messages includes all future (re)definitions in the application.

Data transformation expression named size denotes the number of messages. Notice that size will always be up-to-date with relation to the contents of messages. The **foreach** expression iterates (c.f. `foldl`) over messages with an accumulator y whose initial value is `0`, and on each iteration, the value of y is increased by `1`. We abstract the insertion of a new message into messages with function post where parameter a denotes the author, and m the message.

Notice that the definition of function post uses name size to assign a fresh *id*, which is always up-to-date with relation to the contents of messages. In order to define the behavior of the "thumbs-up" icon we define function like, that increments the *likes* counter of message with identifier i (given by parameter).

State variable user denotes the current logged in user. At this stage, we abstract the different levels of data persistence present in web applications (e.g. sessions, database tables). In this case, messages would be a database table, and user a session variable.

We now gather all the required data, used in the UI, into a collection named wall, containing the message (*m*) and an **action** (*incLikes*) value for each row, representing a thunk containing the identifier of the message in the same row. Using a template language, we can produce a web page definition that corresponds to the UI.

```
def index=
  map(r in wall){
    div {
      img ("/imgs/" + r.m.author + ".jpg")
      r.m.message
      button (img "/imgs/thumbsup.jpg") do r.incLikes
      "(" + r.m.likes + ")"
    }
  }
  div {
    img ("/imgs/" + user + ".jpg")
    textarea#message
    button "Post" do post user #message
  }
```

For the sake of simplicity, we omit styling properties and only present the page skeleton. Expression **map** iterates collection wall, generating a `<div>` element for each message. Declared names can conceivably be accessed as resources through URLs (e.g. name wall can be accessed using a GET request on URL `/wall/`). The execution of operation **do** *r.incLikes*, triggered by clicking the "thumbs-up" icon, is linked to a POST request. Notice that *r.incLikes* denotes an action generated by name like for the respective message. The second **div** expression generates the bottom section of the UI, containing the form with button "Post" linked to the execution of an action. The list of messages in Figure 2 corresponds to accessing name index (e.g. on URL `/`). Whenever the state of the application changes, the web page generated by accessing name index is updated, and the new values are pro-actively pushed to the UI.

In summary, adding messages or clicking a "thumbs-up" icon updates the state variable messages, causes a propagation of changes to names that depend on it (i.e. size, wall and index), thus refreshing them. The propagation paths are detailed in the graph of Figure 1. This dependency graph is incrementally built along

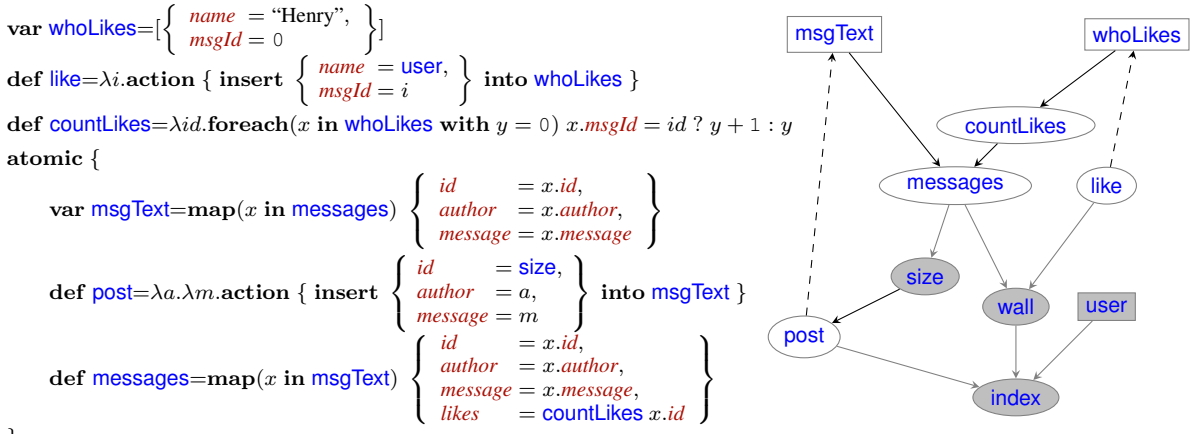

Figure 3: Bulletin Board Reconfiguration.

with each operation, and kept up-to-date with relation to dependencies between data transformation expressions. Nodes correspond to named elements (square: **var**, ellipse: **def**). Solid edges denote dependencies between names, representing the propagation direction. Dashed edges denote delayed action assignments, and are not accounted during propagation. In order to ensure that change propagation does not diverge, we must ensure that the graph is acyclic. This is statically ensured by our type system (Section 3.1), and is one of our final results (Theorem 4.3). Notice that the expressiveness of the language is not limited by the acyclic property of the graph, since the application logic is mainly concentrated in the data transformation expressions, which is a parameter in our setting.

2.2 Evolving the System

Consider a second sprint in the development to add a new feature. Instead of simply counting occurrences, we would like to store which users have liked each message. Using the general scheme of (re)definition of names, one can extend and redefine parts of the running system. The language's type discipline ensures that the typed name dependencies in the system are not broken by each new (re)definition. We also introduce a more flexible evolution mechanism, that combines construction operations into an atomic operation, where transient inconsistencies can happen without impact on execution. The evolution of the new feature in the existing system is attained by the operations in Figure 3, and will change the core of the application without the need to redefine important visible parts (index and wall), depicted in gray on the right-hand side graph.

We start by declaring a new state variable (whoLikes) to store the relation between messages and users. Since there was no previous information of which user liked each message, for the sake of illustrating the structure, we include only one record stating that user "Henry" liked message with *id* `0`. Function like is redefined to insert a record into collection whoLikes instead of incrementing the counter in messages. Auxiliary function countLikes counts the number of "likes" for a particular message (*id*).

The following construction steps target the creation of a new collection (msgText) without the counter, and the redefinition of name messages as a view (c.f. database views) to the pair of state variables (msgText and whoLikes). Since the evolution of the system is interleaved with regular execution (adding messages and "likes"), we next introduce the construction operation **atomic**, that allows to apply a set of operations in a *transactional* style. With this mechanism, we temporarily disallow interaction operations and propagation of changes, which helps avoiding effects such as the one described above (e.g. transient states). In this **atomic** block, we define a new collection to store messages (msgText), without counters, and initialize it by iterating the existing messages. Function post is modified, so to add messages to msgText instead of messages. Finally, messages is redefined as a view to state variables msgText and whoLikes. Notice that messages represents essentially the same information as before, but with a different internal representation that can be used in further extensions.

$$
\begin{aligned}
o &::= r \mid \mathbf{do}\ e \\
r &::= \mathbf{var}\ a{=}e \mid \mathbf{def}\ a{=}e \mid r \circledast r' \\
e &::= a \mid \mathbf{b} \mid x \mid \lambda x.e \mid e\ e' \mid \mathbf{action}\ \{a \leftarrow e\} \mid e\ ?\ e' : e'' \mid e\ \mathsf{op}\ e' \\
&\mid [e_1, \ldots, e_n] \mid \mathbf{foreach}(x\ \mathbf{in}\ e\ \mathbf{with}\ y = e')\ e'' \\
&\mid \mathbf{match}\ e\ \mathbf{with}\ x{::}xs \rightarrow e' \mid [] \rightarrow e''
\end{aligned}
$$

Figure 4: Programming Language Syntax.

In summary, all construction steps are seamlessly applied to the running system, while all elements in the UI (Figure 2), namely index, continue to operate and be refreshed. This allows for a step-by-step construction style where the developer has immediate feedback about the validity and effect of the introduced changes.

3. PROGRAMMING LANGUAGE

We now present our core programming language, whose syntax is presented in Figure 4, and consists in top-level operations (o), comprising construction (r) and interaction (**do** e). These operations rely on a λ-calculus-based core (e), which is a parameter in our model. We define an operational semantics where operations (o) are evaluated with relation to a running application enclosing both state and code. We assume that $a, b, c, \ldots$ range over a set of names $\mathcal{N}$, and $x, y, z, \ldots$ range over a set of variables $\mathcal{V}$. For the sake of simplicity, names are global in the application domain. A modular structure with nested names can be extrapolated from this language, but with no real immediate benefit to the focus of this work. Construction operations (r) include the declaration of state variables (**var** $a{=}e$) in the application's namespace, imperatively associating name a to the value denoted by expression e. Construction operation for the declaration of pure data transformation elements (**def** $a{=}e$) associates name a to the value denoted by expression e. Expressions defining names may use names previously defined. Finally, we introduce the composition operation ($r \circledast r'$) that allows to combine construction operations. In the example from Section 2.2, this was introduced as the **atomic** block. These blocks are evaluated in a *transactional* style, allowing for transient inconsistencies in the system. Interaction operations (**do** e) represent the explicit execution of an action denoted by expression e, which updates the state of the application. Data updates are implicitly and automatically propagated through data transformation elements, thus updating their computed values (cf. a data-driven semantics).

$$\frac{\tau = \lfloor \Delta(a) \rfloor \quad \tau \prec \delta(a)}{\Delta; \Gamma \vdash^{\delta} a : \tau}\ \text{(T-Name)} \qquad \frac{\Delta; \Gamma, x{:}\tau \vdash^{\delta} e : \tau'}{\Delta; \Gamma \vdash^{\delta'} \lambda x{:}\tau.e : \tau \xrightarrow{\delta} \tau'}\ \text{(T-Abs)}$$

$$\frac{\Delta; \Gamma \vdash^{\delta} e : \tau \xrightarrow{\delta'} \tau' \quad \Delta; \Gamma \vdash^{\delta} e' : \tau \quad \delta' \sqsubseteq \delta}{\Delta; \Gamma \vdash^{\delta} e\ e' : \tau'}\ \text{(T-App)}$$

$$\frac{\Delta, a{:}\mathsf{var}_{\delta}(\tau); \Gamma \vdash^{\delta[a:\mathsf{var}(\tau)]} e : \tau}{\Delta, a{:}\mathsf{var}_{\delta}(\tau); \Gamma \vdash^{\delta'} \mathbf{action}\ \{a \leftarrow e\} : \mathsf{Action}(\delta[a{:}\mathsf{var}(\tau)])}\ \text{(T-Action)}$$

Figure 5: Typing Rules for Expressions.

The functional core of the language (e) includes names (a) that denote the corresponding values in the application state, basic values $\mathbf{b}$ (e.g. **true**, **false**), and variables x. Abstraction ($\lambda x.e$) and application ($e\ e'$) follow a call-by-value semantics. The expression $\mathbf{action}\ \{a \leftarrow e\}$ contains a delayed assignment to state variable a. We also include the ternary conditional operator (?:), and for the sake of simplicity, assume a set of binary operators (op) over basic values. We extend the base λ-calculus with collections, with a constructor $[e_1, ..., e_n]$, and corresponding concatenation (@) and append (::) operators (in op). Iterator $\mathbf{foreach}(x\ \mathbf{in}\ e\ \mathbf{with}\ y = e')\ e''$ denotes a fold-left operation on the collection denoted by expression e, and the iterated expression e''. Variable x denotes the current element in the collection, and variable y denotes either the value of expression e'' in the previous iteration, or the initial value given by expression e' in the first iteration. Collections destructor $\mathbf{match}\ e\ \mathbf{with}\ x{::}xs \rightarrow e' \mid [] \rightarrow e''$ denotes one of two cases, depending on the value of expression e. In the case of a non-empty collection, the evaluation proceeds with expression e', where variable x denotes the head of the collection, and xs denotes its tail. For an empty collection ($[]$), the result is denoted by expression e''.

In Section 2, we used a slightly more elaborate syntax to write the example that can be directly mapped into the syntax of Figure 4.

$$\mathbf{insert}\ e\ \mathbf{into}\ \mathsf{a} \triangleq \mathsf{a} \leftarrow \mathsf{a}@[e]$$
$$\mathbf{update}\ x\ \mathbf{in}\ \mathsf{a}\ \mathbf{with}\ e\ \mathbf{where}\ e' \triangleq \mathsf{a} \leftarrow \mathbf{foreach}(x\ \mathbf{in}\ \mathsf{a}\ \mathbf{with}\ y = [])\ y@[e'\ ?\ e : x]$$
$$\mathbf{map}(x\ \mathbf{in}\ e)\ e' \triangleq \mathbf{foreach}(x\ \mathbf{in}\ e\ \mathbf{with}\ y = [])\ y@[e']$$
$$\mathbf{atomic}\ \{r_1, \ldots, r_n\} \triangleq r_1 \circledast \ldots \circledast r_n$$

We also used a template language to produce web pages that can extend the language orthogonally. We next present the type system and operational semantics. Since the construction and execution of applications are tightly intertwined, we start by describing the type system, and then define the operational semantics.

3.1 Type System

We now define a type and effect system to discipline the construction of systems so that soundness of the type system implies that the application is safe and responsive. To do so, we statically keep track of name dependencies to avoid the creation of unguarded cyclic dependencies. We say that a dependency cycle is guarded if it crosses an action value, and hence needs an explicit interaction operation to be activated. Any unguarded cycle would cause the propagation process to diverge. The absence of runtime errors and infinite propagation cycles are implied by the standard progress and preservation results (Section 4). Our type language is defined by

$$\tau ::= \beta(\mathbf{b}) \mid \tau^* \mid \tau \xrightarrow{\delta} \tau' \mid \mathsf{Action}(\delta)$$

where we include a set of basic types (e.g. Bool) with $\beta(\mathbf{b})$ denoting the type of basic value $\mathbf{b}$, types for homogeneous collections (τ^*), functions ($\tau \xrightarrow{\delta} \tau'$), and actions ($\mathsf{Action}(\delta)$). Function and action types, specify the behavior of delayed expressions, and use δ to denote a set of typed dependencies on declared names. This bookkeeping of dependencies is crucial to determine the soundness of a name's (re)definition (described in detail in Section 3.3). Typed dependencies sets are defined by

$$\delta ::= a : \tau, \delta \mid a : \mathsf{var}(\tau), \delta \mid \varepsilon$$

and capture information about the type of a used name, and whether it denotes a state variable. A typed dependency of the form $a{:}\tau$ means that name a is used with type τ, while the form $a{:}\mathsf{var}(\tau)$ means that name a is used as state variable with type τ. Our type and effect system is defined by two layers of typing judgments that target operations and expressions.

$$\Delta \vdash o : \Delta' \quad \text{(Operations)} \qquad \Delta; \Gamma \vdash^{\delta} e : \tau \quad \text{(Expressions)}$$

Typing environments Δ register the type and usage information of names. Expressions type environments Γ map variables $x, y, z, ...$ to types. The typing judgment for operations registers the effect (Δ') of operation o, and the judgment for expressions includes the typed dependencies (δ) of expression e. Typing environments Δ map names to type annotations (σ), which are designed to carry information about the names' denotation, and are defined by

$$\sigma ::= \mathsf{var}_{\delta}(\tau) \mid \mathsf{def}_{\delta}(\tau)$$

where δ denotes a typed dependencies set (as defined above), to denote all typed dependencies of the expression associated the annotated name (i.e. the direct dependencies). Typed dependency sets are essential to avoid unguarded cyclic definitions, which would cause infinite propagation loops. In Section 3.3, we show how to statically satisfy and preserve this. Consider the following auxiliary definitions essential to understand the typing rules for expressions.

Definition 3.1 (Type of Annotation). *We obtain the type of an annotation σ, as follows*

$$\lfloor \mathsf{var}_{\delta}(\tau) \rfloor \triangleq \tau \qquad \lfloor \mathsf{def}_{\delta}(\tau) \rfloor \triangleq \tau$$

Definition 3.2 (Typed Dependency Coercion). *We define the type dependency coercion, as follows*

$$\tau \prec \tau \qquad \tau \prec \mathsf{var}(\tau) \qquad \mathsf{var}(\tau) \prec \mathsf{var}(\tau)$$

The former allows to extract the actual type from a type annotation, and the latter defines a coercion mechanism. Notably, we define that a *write* typed dependency ($\mathsf{var}(\tau)$) can be used whenever a simple dependency (τ) is required. This relation can also be lifted to define a relation between typed dependency sets $\delta \sqsubseteq \delta'$.

The typing relation on expressions is inductively defined, and follows standard lines, except that they are extended with the invariant conditions about typed dependencies (δ). We show, in Figure 5, only the most interesting cases. In the typing relation on expressions, all language values ($\mathbf{b}$, x, $\lambda x.e$ and $\mathbf{action}\ \{a \leftarrow e\}$) are typed with all typed dependencies sets. In the case of rule T-Name, a name a is typed with relation to an environment and a typed dependencies set that agree in the type of a, modulo the typed dependency coercion of Definition 3.2, and the type of the annotation (Definition 3.1). Abstraction $\lambda x.e$ is typed (T-Abs) with any typed dependency set δ', given that the resulting function type is annotated with the typed dependency set (δ) that types the abstraction body e. This captures the dependencies set of a closure, which conservatively signal the typed dependencies of the abstraction body, and defer its use to the time of its application. That can be observed in rule T-App, where the typed dependencies set of the functional type (δ') is required to be a subset (or equal) of the application typed dependencies set ($\delta' \sqsubseteq \delta$). Rule T-Action requires that in an assignment, the type of the expression (e) agrees with the type of the annotated name (a), and that name is a state variable ($\mathsf{var}_{\delta}(\tau)$). Expression e can refer to the old value of the assigned variable, hence its typed dependencies set is $\delta[a{:}\mathsf{var}(\tau)]$. The resulting action type keeps track of the use of name a as a state variable ($a{:}\mathsf{var}(\tau)$), therefore blocking all code changes that try to redefine it into a data transformation expression ($\mathbf{def}$).

The type and effect system on operations is defined by a judgment of the form $\Delta \vdash o : \Delta'$ asserting that operation o produces the

$$\frac{\Delta;\varepsilon \overset{\delta}{\vdash} e:\tau \quad a \notin \mathsf{dom}(\delta)}{\Delta \vdash \mathbf{var}\ a{=}e : \{a : \mathsf{var}_\delta(\tau)\}}\ (\textsc{TVar}) \qquad \frac{\Delta;\varepsilon \overset{\delta}{\vdash} e:\tau \quad a \notin \mathsf{dom}(\delta)}{\Delta \vdash \mathbf{def}\ a{=}e : \{a : \mathsf{def}_\delta(\tau)\}}\ (\textsc{TDef}) \qquad \frac{\Delta \vdash r : \Delta' \quad \Delta \uplus \Delta' \vdash r' : \Delta''}{\Delta \vdash r \circledast r' : \Delta' \uplus \Delta''}\ (\textsc{TComp}) \qquad \frac{\Delta;\varepsilon \overset{\delta'}{\vdash} e : \mathsf{Action}(\delta)}{\Delta \vdash \mathbf{do}\ e : \varepsilon}\ (\textsc{TDo})$$

Figure 6: Typing Rules for Operations.

effect Δ' with relation to the typing environment Δ. The typing based on effects of operations, inductively defined in Figure 6, allows us to capture the incremental effects of each operation, and therefore verify that a given operation is compatible with the running system (further detailed in Section 3.3 with Definition 3.9). Notice that the type verification based on the effects is essential to dynamically accept new construction operations, by allowing to predict if the system is sound after executing all steps that are in the evaluation queue to be processed. We use the following auxiliary definition about the right-biased union of two typing environments.

Definition 3.3 (Typing Environment Union). *The union of typing environments Δ and Δ', written $\Delta \uplus \Delta'$, is defined by*

$$\Delta \uplus \Delta' = \{a : \Delta(a) \mid a \in \mathsf{dom}(\Delta) - \mathsf{dom}(\Delta')\} \cup \Delta'$$

We next define the notion of well-formed type and well-formed typed dependencies. The first (Definition 3.4) asserts that a given type is well-formed with relation to a typing environment Δ. The most interesting cases are for function and action types, where we also assert that the typed dependencies (δ) are well-formed, according to Definition 3.5. Regarding Definition 3.5, it asserts that all typed dependencies agree with the type of the name, and also that type associated with the name is well-formed (Definition 3.4).

Definition 3.4 (Well-Formed Type). *A type τ is well-formed wrt. typing environment Δ, if $\Delta \vdash \tau$ is derivable by*

$$\Delta \vdash \beta(\mathbf{b}) \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \tau^*} \qquad \frac{\Delta \vdash \delta}{\Delta \vdash \mathsf{Action}(\delta)} \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash \tau' \quad \Delta \vdash \delta}{\Delta \vdash \tau \xrightarrow{\delta} \tau'}$$

Definition 3.5 (Well-Formed Typed Dependencies). *A typed dependencies set δ is well-formed with relation to a typing environment Δ, if $\Delta \vdash \delta$ can be derived by*

$$\Delta \vdash \varepsilon$$

$$\frac{\lfloor \Delta(a) \rfloor \prec \tau \quad \Delta \vdash \tau \quad \Delta \vdash \delta'}{\Delta \vdash \delta', a : \tau} \qquad \frac{\Delta(a) = \mathsf{var}_\delta(\tau) \quad \Delta \vdash \tau \quad \Delta \vdash \delta'}{\Delta \vdash \delta', a : \mathsf{var}(\tau)}$$

The effect of the two construction operations **var** $a{=}e$ and **def** $a{=}e$ is specified by rules TVar and TDef (Figure 6). The expressions e typed dependencies (δ) and type (τ) are included in the registered effect ($\mathsf{def}_\delta(\tau)$ or $\mathsf{var}_\delta(\tau)$ respectively). Notice that a definition for name a cannot directly depend on the name itself (represented by $a \notin \mathsf{dom}(\delta)$). In a composition operation (TComp), the left operand is typed with relation to the initial typing environment Δ, while the right operand is typed in a typing environment obtained by combining the initial environment with the effect of the left-hand side operation ($\Delta \uplus \Delta'$). The effect of the composition is the combined effects of both operands ($\Delta' \uplus \Delta''$). Interaction operations (**do** e) do not produce any effects (ε), i.e. the application definition is not modified, only the state is modified at runtime.

3.2 Operational Semantics

We define a reactive operational semantics by means of two layers of small-step reduction relations (for construction operations and for expressions). The semantics of construction operations is defined on runtime configurations ($\Delta; \mathcal{S}; \mathcal{Q}$) representing a complete live system. Typing environment Δ describes the current system type specification, $\mathcal{S}$ is a state mapping names (a) to tuples with the form (e, ν), with e an expression, whose free names range over the domain of $\mathcal{S}$, and ν the current denotation of the name a, that can be either a computed value (v), or the special denotation *undefined* ($\square$). We write $\mathcal{S}[a \mapsto (e, \nu)]$ to denote a state obtained from state $\mathcal{S}$ and mapping a to (e, ν). Runtime values are defined by

$$v ::= \mathbf{b} \mid \lambda x.e \mid \mathbf{action}\ \{a \leftarrow e\} \mid [v_0, \ldots, v_n]$$

and includes base values $\mathbf{b}$ (e.g. $\mathbf{true}$, $\mathbf{false}$), abstractions, actions, and collections of values. The third element of a runtime configuration, $\mathcal{Q}$, denotes a queue of operations, and is defined by

$$\begin{array}{lcl} \mathcal{Q} & ::= & \mathbf{do}\ e; \mathcal{Q} \mid a; \mathcal{Q} \mid a\langle e\rangle; \mathcal{Q} \mid [q]^{\Delta}_{\mathcal{S}}; \mathcal{Q} \mid \varepsilon \\ q & ::= & r; q \mid a; q \mid a\langle e\rangle; q \mid \varepsilon \end{array}$$

The queue disciplines the evaluation of a set of related events. Queued events have one of the following sorts: interaction operations (**do** e), names to be refreshed (a), names currently being refreshed ($a\langle e\rangle$), or construction blocks ($[q]^{\Delta}_{\mathcal{S}}$). Construction blocks are annotated with a typing environment Δ and a state $\mathcal{S}$, that describe its partial effect. A construction block is defined over a queue of selected events q, that exclude the execution of actions (**do** e). Blocks can contain **var** $a{=}e$ or **def** $a{=}e$ construction operations (r), names queued to be refreshed (a), and names being evaluated ($a\langle e\rangle$).

Consider some extra definitions, auxiliary in the reduction relation on operations. We define the subscribers of a name a, as all data transformation expressions (**def**) that depend directly on name a. This definition is crucial to setup the propagation process, that follow the data transformation expressions and does not go into expressions that initialize state variables.

Definition 3.6 (Subscribers). *The set of subscribers of name a, with relation to Δ, written $\mathsf{subscribers}_\Delta(a)$, is defined by*

$$\mathsf{subscribers}_\Delta(a) = \{\ b \mid \Delta(b) = \mathsf{def}_\delta(\tau) \wedge a \in \mathsf{dom}(\delta)\ \}$$

We also define the union of two states ($\mathcal{S} \uplus \mathcal{S}$) as the right-biased union of two states (similar to Definition 3.3 on typing environments). Given these auxiliary definitions, the reduction relation on operations is inductively defined in Figure 7, following the general structure $\Delta; \mathcal{S}; \mathcal{Q} \rightarrow \Delta'; \mathcal{S}'; \mathcal{Q}'$ where operations are executed according to the discipline imposed by queue $\mathcal{Q}$, modifying the typing environment Δ and state $\mathcal{S}$ accordingly. We divide the description of the operational semantics, and start by the rules that specify the handling of queued events. Section 3.3 introduces runtime rules that allow to insert operations into the queue.

Queued construction operations reduce through rules S-RVar and S-RDef. The partial typing environment (Δ') is updated with the effect of the operation (Δ''), and the partial state ($\mathcal{S}'$) is updated to associate name a to the defined expression e, and an undefined value ($\square$). The name a is placed in the queue to be evaluated. This is the general schema to evaluate expressions, through the queue, and with precedence towards operations already in the queue. The evaluation of the expression associated to a name is specified by rules S-Queue and S-RQueue that reduces the event a in the queue to the event $a\langle e\rangle$, according to the current state. Rule S-Queue works with relation to $\mathcal{S}$, and rule S-RQueue works on the partially constructed state ($\mathcal{S} \uplus \mathcal{S}'$). The remaining pair of rules work according to the same convention over states and partial states. Rule S-Step (S-RStep) reduces expression e associated to a name a in the queue. This step relies on the reduction relation for expressions defined below. The evaluation of a terminated queued name (i.e. it refers to a value) is reduced by rule S-Value (S-RValue) that updates the state, replacing the old value v' (ν) associated to name a by the new value v. In the case of a value update outside a construction block (S-Value), the subscribers of the updated name (Definition 3.6) are placed at the beginning of the queue. This ensures that we immediately propagate changes into other names, thus giving the reactive behavior to our semantics. In the case of a construction block,

$$\frac{\Delta \uplus \Delta' \vdash \mathbf{var}\ a{=}e : \Delta''}{\Delta; \mathcal{S}; ([\mathbf{var}\ a{=}e; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta; \mathcal{S}; ([a; q]^{\Delta' \uplus \Delta''}_{\mathcal{S}'[a \mapsto (e, \square)]}; \mathcal{Q})}\ \text{(S-RVAR)} \qquad \frac{\mathcal{S}(a) = (e, v)}{\Delta; \mathcal{S}; (a; \mathcal{Q}) \to \Delta; \mathcal{S}; (a\langle e\rangle; \mathcal{Q})}\ \text{(S-QUEUE)} \qquad \frac{(\mathcal{S} \uplus \mathcal{S}')(a) = (e, \nu)}{\Delta; \mathcal{S}; ([a; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta; \mathcal{S}; ([a\langle e\rangle; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q})}\ \text{(S-RQUEUE)}$$

$$\frac{\Delta \uplus \Delta' \vdash \mathbf{def}\ a{=}e : \Delta''}{\Delta; \mathcal{S}; ([\mathbf{def}\ a{=}e; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta; \mathcal{S}; ([a; q]^{\Delta' \uplus \Delta''}_{\mathcal{S}'[a \mapsto (e, \square)]}; \mathcal{Q})}\ \text{(S-RDEF)} \qquad \frac{\mathcal{S}; e \to e'}{\Delta; \mathcal{S}; (a\langle e\rangle; \mathcal{Q}) \to \Delta; \mathcal{S}; (a\langle e'\rangle; \mathcal{Q})}\ \text{(S-STEP)} \qquad \frac{\mathcal{S} \uplus \mathcal{S}'; e \to e'}{\Delta; \mathcal{S}; ([a\langle e\rangle; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta; \mathcal{S}; ([a\langle e'\rangle; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q})}\ \text{(S-RSTEP)}$$

$$\frac{s = \mathsf{subscribers}_{\Delta}(a)}{\Delta; \mathcal{S}[a \mapsto (e, v')]; (a\langle v\rangle; \mathcal{Q}) \to \Delta; \mathcal{S}[a \mapsto (e, v)]; (s; \mathcal{Q})}\ \text{(S-VALUE)} \qquad \frac{(\mathcal{S} \uplus \mathcal{S}')(a) = (e, \nu)}{\Delta; \mathcal{S}; ([a\langle v\rangle; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta; \mathcal{S}; ([q]^{\Delta'}_{\mathcal{S}'[a \mapsto (e, v)]}; \mathcal{Q})}\ \text{(S-RVALUE)} \qquad \frac{\mathcal{S}; e \to e'}{\Delta; \mathcal{S}; (\mathbf{do}\ e; \mathcal{Q}) \to \Delta; \mathcal{S}; (\mathbf{do}\ e'; \mathcal{Q})}\ \text{(S-DO)}$$

$$\frac{s = \mathsf{subscribers}_{\Delta \uplus \Delta'}(\mathsf{dom}(\Delta'))}{\Delta; \mathcal{S}; ([\varepsilon]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q}) \to \Delta \uplus \Delta'; \mathcal{S} \uplus \mathcal{S}'; (s; \mathcal{Q})}\ \text{(S-REMPTY)} \qquad \Delta; \mathcal{S}[a \mapsto (e', v)]; (\mathbf{do\ action}\ \{a \leftarrow e\}; \mathcal{Q}) \to \Delta; \mathcal{S}[a \mapsto (e, v)]; (a; \mathcal{Q})\ \text{(S-DO-ACTION)}$$

Figure 7: Operations Operational Semantics.

this procedure is delayed until the end of the block, and names are placed in bulk in the queue (rule S-REMPTY). Rule S-REMPTY uses a lifted definition of the subscribers set (Definition 3.6) for sets of names. At the end of a construction block (S-REMPTY) the partially computed typing environment and state are combined with the main typing environment and state, as if committing changes. This is a key point, where our static type analysis ensures that the soundness of the system (that can be transient in a construction block) is restored, and propagation proceed.

Finally, rules S-DO and S-DO-ACTION handle the interaction operation ($\mathbf{do}\ e$). Rule S-DO reduces the inner expression e, based on the reduction relation for expressions. The reduction of an action value (S-DO-ACTION) proceeds by updating the expression (e') associated to the assigned name (a) in the state. The general evaluation strategy then follows; name a is added to the queue, evaluated and the state is updated (rules S-STEP and S-VALUE respectively).

The reduction relation on configurations depends on a reduction relation for expressions with relation to a state. We follow the structure $\mathcal{S}; e \to e'$ that specifies that expression e reduces to e' with relation to state $\mathcal{S}$. Reduction of expressions does not change the state and follows standard lines, and is therefore omitted This description concludes the operational semantics regarding the execution of the system. We now describe how a system may evolve by adding new construction operations to the queue.

3.3 Incremental Construction

We next describe how our runtime system dynamically evolves by combining the presented type system (Section 3.1) and operational semantics (Section 3.2). For that purpose, first consider some auxiliary definitions that enable our runtime system to dynamically check conditions that ensure the soundness of the system.

DEFINITION 3.7 (TYPED DEPENDENCIES EXPANSION). *The expansion of typed dependencies δ, in a given names environment Δ, written $\sqcup_{\Delta}(\delta)$, is defined by*

$$\sqcup_{\Delta}(\delta) \triangleq \mathsf{dom}(\delta) \cup \bigcup_{a \in \mathsf{dom}(\delta)} \{\ \sqcup_{\Delta}(\delta') \mid \Delta(a) = \mathsf{def}_{\delta'}(\tau)\ \}$$

Computing the expansion of the typed dependencies is essential to statically avoid circular dependencies, and ensuring that the propagation of changes does not diverge. Notice that state variables in δ are not expanded, this is sound and corresponds with the restriction on change propagation into state variables. One important invariant on runtime configurations is the absence of unguarded dependency cycles. We next define a (static) property on typing environments, that directly implies this invariant.

DEFINITION 3.8 (ACYCLIC). *A typing environment Δ is acyclic if, $\Delta \downarrow$ is derivable by the rules*

$$\frac{\forall a \in \mathsf{dom}(\Delta).\Delta \downarrow_a}{\Delta\downarrow} \qquad \frac{\Delta(a) = \mathsf{var}_{\delta}(\tau) \qquad \Delta \vdash \delta, a : \tau}{\Delta\downarrow_a}$$

$$\frac{\Delta(a) = \mathsf{def}_{\delta}(\tau) \qquad a \notin \sqcup_{\Delta}(\delta) \qquad \Delta \vdash \delta, a : \tau}{\Delta\downarrow_a}$$

We define the notion of compatibility between two typing environments Δ and Δ' in Definition 3.9.

DEFINITION 3.9 (COMPATIBLE). *We define compatibility of typing environments Δ and Δ', written $\Delta \vDash \Delta'$, if $\Delta \uplus \Delta'\downarrow$.*

Verifying that a typing environment does not contain unguarded cycles, and that two typing environments are compatible is a crucial step when adding operations to a running system. The notion of compatibility ensures that for the combined typing environment $\Delta \uplus \Delta'$ (Definition 3.3), all names are acyclic ($a \notin \sqcup_{\Delta}(\delta)$), and their typed dependencies are well-formed ($\Delta \vdash \delta, a : \tau$) according to Definition 3.5. This enables us to guarantee that either type changes, or a name redefined from a state variable into a pure data transformation element (or vice-versa) does not break type safety.

Since our runtime queue may contain unprocessed construction blocks, we define the notion of queue effects (Definition 3.10), that computes the effects of an unprocessed queue, and allows to add new construction operations to the queue, in a verified way.

DEFINITION 3.10 (QUEUE EFFECTS). *We define the effects of a queue $\mathcal{Q}$, wrt. a typing environment Δ, written $\Delta \vdash \mathcal{Q} \dashv \Delta'$, producing the effects typing environment Δ', as follows*

$$\frac{\Delta \vdash \mathcal{Q} \dashv \Delta'}{\Delta \vdash a; \mathcal{Q} \dashv \Delta'} \qquad \frac{\Delta \vdash [q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta''}{\Delta \vdash [a; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta''} \qquad \frac{\Delta \uplus \Delta' \vdash \mathcal{Q} \dashv \Delta''}{\Delta \vdash [\varepsilon]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta' \uplus \Delta''}$$

$$\frac{\Delta \vdash \mathcal{Q} \dashv \Delta'}{\Delta \vdash a\langle e\rangle; \mathcal{Q} \dashv \Delta'} \qquad \frac{\Delta \uplus \Delta' \vdash r : \Delta'' \quad \Delta \uplus \Delta' \uplus \Delta'' \vdash [q]^{\varepsilon}_{\varepsilon}; \mathcal{Q} \dashv \Delta'''}{\Delta \vdash [r; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta' \uplus \Delta'' \uplus \Delta'''}$$

$$\frac{\Delta \vdash [q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta''}{\Delta \vdash [a\langle e\rangle; q]^{\Delta'}_{\mathcal{S}'}; \mathcal{Q} \dashv \Delta''} \qquad \frac{}{\Delta \vdash \varepsilon \dashv \varepsilon} \qquad \frac{\Delta \vdash \mathbf{do}\ e : \varepsilon \quad \Delta \vdash \mathcal{Q} \dashv \Delta'}{\Delta \vdash \mathbf{do}\ e; \mathcal{Q} \dashv \Delta'}$$

With these auxiliary definitions, we now introduce how interaction operations are added to the running application. The following rule enables the addition of an operation $\mathbf{do}\ e$ to the end of the queue

$$\frac{\Delta \vdash \mathcal{Q} \dashv \Delta' \qquad \Delta \uplus \Delta' \vdash \mathbf{do}\ e : \varepsilon}{\Delta; \mathcal{S}; \mathcal{Q} \xrightarrow{\mathbf{do}\ e} \Delta; \mathcal{S}; (\mathcal{Q}; \mathbf{do}\ e)}\ \text{(INTERACTION)}$$

Notice that the operation $\mathbf{do}\ e$ is typed under the typing environment $\Delta \uplus \Delta'$, corresponding to the combination of the current system typing environment Δ with the effects of the queue Δ'. This ensures that the new operation takes into account all pending events still queued. Recall that interactions operations do not change the system's definition, hence the resulting typing environment ε. Next, we present the rule that enqueues a construction operation r at the end of the queue

$$\frac{\Delta \vdash \mathcal{Q} \dashv \Delta' \quad \Delta \uplus \Delta' \vdash r : \Delta'' \quad \Delta \uplus \Delta' \vDash \Delta''}{\Delta; \mathcal{S}; \mathcal{Q} \xrightarrow{r} \Delta; \mathcal{S}; (\mathcal{Q}; [\![r]\!]^{\varepsilon}_{\varepsilon})}\ \text{(CONSTRUCTION)}$$

Similarly, we gather the remaining queue effects Δ', and the new operation r is typed under the combined typing environment $\Delta \uplus \Delta'$. Since, in this case, we are reconfiguring the system, we must check that the new operation is compatible with the current running system ($\Delta \uplus \Delta' \vDash \Delta''$). This ensures that the queued operation r

will not break type safety, or cause propagation of changes to diverge. Notice that the construction block initial typing environment and state are both empty (ε). These will later be constructed by the operational semantics rules that reduce the construction blocks.

With the notion of compatibility (Definition 3.9), we allow for a composition construction operation to introduce transient inconsistencies in-between *sub*-operations (e.g. cyclic definitions). Our composition operator enables the (re)definition of several names that may introduce transient inconsistencies, while ensuring that in the end the application is sound. In Section 4 we present our main results that include type safety, and the convergence of propagation.

4. TYPE SAFETY

We next present the main soundness results for our language, that are based on proving safety (progress and preservation), and convergence of the propagation of changes. Besides common typing errors, our system also statically ensures that the intended reactive behavior is kept sound even when the system evolves. The results presented here follow the syntactic approach of [13]. We state our main results and provide full proofs in a technical report [6].

We next present our runtime progress result (Theorem 4.1) and type preservation (Theorem 4.1). In the first, we ensure a stronger invariant of the reduction relation, states that all names in the current state have a value associated ($\mathcal{S}_\mathsf{v}(a) = v$).

THEOREM 4.1 (RUNTIME PROGRESS). *For all runtime configurations* $(\Delta; \mathcal{S}; \mathcal{Q})$*, if* $\vdash (\Delta; \mathcal{S}; \mathcal{Q})$ *and* $\mathcal{Q} \neq \varepsilon$ *and* $\forall a \in \mathsf{dom}(\mathcal{S})$. $\mathcal{S}_\mathsf{v}(a) = v$ *then, there is a program configuration* $(\Delta'; \mathcal{S}'; \mathcal{Q}')$ *such that* $\Delta; \mathcal{S}; \mathcal{Q} \rightarrow \Delta'; \mathcal{S}'; \mathcal{Q}'$ *and* $\forall a \in \mathsf{dom}(\mathcal{S}').\mathcal{S}'_\mathsf{v}(a) = v$.

THEOREM 4.2 (RUNTIME TYPE PRESERVATION). *For all runtime configurations* $(\Delta; \mathcal{S}; \mathcal{Q})$ *and* $(\Delta'; \mathcal{S}'; \mathcal{Q}')$*, if* $\vdash \Delta; \mathcal{S}; \mathcal{Q}$*, and* $\Delta; \mathcal{S}; \mathcal{Q} \rightarrow \Delta'; \mathcal{S}'; \mathcal{Q}'$ *then,* $\vdash \Delta'; \mathcal{S}'; \mathcal{Q}'$.

Based on a strong normalizing result for the simply typed λ-calculus (c.f. [12]), we are able to establish an even stronger result, where all well-typed runtime configurations with a non-empty queue, reach a runtime configuration with an empty queue after a finite number of steps.

THEOREM 4.3 (CONVERGENCE). *For all runtime configurations* $(\Delta; \mathcal{S}; \mathcal{Q})$*, if* $\vdash (\Delta; \mathcal{S}; \mathcal{Q})$ *then* $\Delta; \mathcal{S}; \mathcal{Q} \rightarrow^* \Delta'; \mathcal{S}'; \varepsilon$.

Notice that this result is based (and parametric) on the termination of the functional core, which in our case is a simply typed λ-calculus extended with collections. Using any expression language free of side-effects, for which we can prove termination (e.g. [10]), then the result also holds. These results allow us to ensure that the automatic propagation of changes converges, and dynamic reconfiguration of both code and data is always sound.

5. RELATED WORK

Several works on dynamic reconfiguration and incremental computation of software systems have already been proposed [7, 11]. These solutions are mostly designed for imperative programming languages. For instance, DSU [7] provides dynamic updates in programs by explicitly defining points where updates may occur. Up to some point our approach is similar, however we use the same uniform mechanism that allows to build and evolve applications.

AFP [2] uses an underlying dependency graph and a change propagation algorithm to adapt the output when input changes. We use a similar technical approach, where propagation is also encoded into operational semantics by the queue. Our approach also goes further by providing a reactive framework that allows to safely dynamically reconfigure and evolve applications. Several Functional Reactive Programming approaches have been proposed [9, 5]. However, neither of these approaches combines reactivity with a statically verified evolution mechanism as the one presented in this paper. In FrTime [5], reactivity is embedded into a call-by-value dynamic programming language. In principle, an untyped fragment of our language can be encoded into FrTime. However, in such untyped fragment it would be possible to define programs with transient inconsistent states causing the propagation to diverge. In our approach, we ensure that the propagation of changes does not diverge, and that all (re)definitions are safe, thus making it impossible to define infinite propagation cycles.

TouchDevelop [4] provides an environment with immediate feedback in the development of interfaces and small applications. The edit-compile-run cycle is tightened by allowing the display code to be refreshed without restarting programs. Although related in their goals, we target the development of data-centric applications where the correct evolution of state is of major importance.

6. FINAL REMARKS

We have introduced a core language, its type system and operational semantics, that is suitable for safe and incremental construction of live software systems. Our language is the first approach that is uniform, statically type safe, and supports the dynamic evolution of both code and data. Our main results include the soundness of the whole system, and the convergence of the propagation. We also presented a working prototype [1] that allows to build and evolve web applications using the techniques presented in this paper.

We identify some future challenges in our model, both in the formal and pragmatic domains. On the pragmatic side, we may extend the language expressiveness by parameterizing the model with a full-fledged functional or imperative language. All these extensions are being pursued by developments in our prototype [1]. On the formal side, extensions include the parallel scheduling of the queue and corresponding type discipline based on notions of separation.

Acknowledgments.. We thank the anonymous reviewers for their insightful comments and suggestions. This work is FCT/MEC grant SFRH/BD/73249/2010.

7. REFERENCES

[1] Prototype. http://tiny.cc/prototype.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive Functional Programming. *ACM TOPLAS*, 28(6), 2006.

[3] P. Bhattacharya and I. Neamtiu. Dynamic Updates for Web and Cloud Applications. In *Proceedings of APLWACA*, 2010.

[4] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of PLDI*, 2013.

[5] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of ESOP*, 2006.

[6] M. Domingues and J. C. Seco. Type Safe Evolution of Live Systems. Technical report, NOVA-LINCS – UNL, 2015. http://miguel.domingues.pt.

[7] M. Hicks and S. Nettles. Dynamic Software updating. *ACM TOPLAS*, 27(6), 2005.

[8] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of POPL*, 2011.

[9] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of PPDP*, 2005.

[10] D. McAllester and K. Arkoudas. Walther recursion. In *Automated Deduction—Cade-13*. 1996.

[11] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM TOPLAS*, 29(4), 2007.

[12] W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2), 1967.

[13] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1994.

[14] D. Yellin and R. Strom. INC: a language for incremental computations. In *Proceedings of PLDI*, 1988.