Typeful Updates on Reactive Live Web Programming*

Miguel Domingues and João Costa Seco

CITI and Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Abstract. We introduce a core reactive and imperative programming language for data-centric applications, that supports dynamic upgrades of both code and data. We use a monadic structure to control the automatic propagation of changes through a data-flow graph, thus allowing for imperative constructs to be safely used in a reactive context.
Our approach enables the construction of rich and reactive user interfaces, while supporting a core and uniform reconfiguration mechanism for both application logic and persistent state. This is well-suited for designing new development tools to support the agile deployment of web applications. We provide a flexible programming style, where applications are incrementally built and maintained by sequences of verified construction operations, on live instances of software systems.
We present an operational semantics and a monadic type system that ensure the soundness of change propagation through the data-flow graph, which includes a convergence result and the correctness of dynamic update of graph dependencies in the presence of reconfiguration actions.

1 Introduction

Agile development frameworks and fast deployment methodologies are gaining popularity, especially in the domain of web applications. However, these techniques often require a large amount of time dedicated to redefine and refactor existing code. Integrated platforms and smart programming environments give semi-automatic support to soundly modify parts of an application. Nevertheless, they are not providing a real incremental mechanism for software systems. Another challenge that is gaining importance, is how to design software architectures that easily propagate changes occurring in the application persistent data layer and present them in user interfaces [1,5]. Traditional layered and heterogeneous architectures for web applications, usually forces this reactive behavior of applications to be defined by ad-hoc, and hand-crafted code. Implementing this kind of reactive behavior is even more complex in the context of distributed and collaborative applications, where several users interact and compete for the same application code and data. In data-centric web applications, users expect their interface to be up-to-date when other users modify some data. Our attention is drawn to the problem of how to make the evolution of data-centric web

* Technical Report (extended version with proofs) - October, 2013

applications less time consuming and error prone, especially when the reactive behavior is explicitly programmed.

We envision a style of incremental programming [6,17,22] that captures a set of core construction operations over a live program instance [5]. Our approach allows for developers to continuously evolve code [7,10,18], and reconfigure the underlying data schema accordingly [2,12]. We introduce a possible instantiation of such programming environment by means of a core reactive and imperative programming language, suitable for building data-centric applications, that supports dynamic reconfiguration of both application code and data.

Our language, defines three essential top-level operations that support the declaration of state variables, pure data transformation expressions, and execution of imperative monadic actions. Intuitively, these elements correspond to an application's persistent state, data querying and manipulation code, and handling of stateful operations associated to user interface events, respectively. The language follows a "push"-reactive operational semantics [11]. Persistent state modifications are implicitly and automatically propagated, updating previously computed values of pure data transformation elements. The application's data-flow graph is shaped by a monadic structure, in such a way that imperative code is separated and change propagation is disciplined. Hence, allowing for imperative constructs to be safely used in a reactive context, and ensuring the absence of infinite propagation loops.

In summary, we provide a flexible style of live programming, where applications are incrementally built and maintained by sequences of verified operations. Upon reconfiguration, the data-flow dependency graph is dynamically updated to ensure that propagation of changes remains sound. To the best of our knowledge, this is the first work presented combining reactivity and imperative constructs, with a statically verified dynamic reconfiguration mechanism.

Our key contributions can be summarized as follows:

- We define a novel core reactive and imperative language, that supports dynamic reconfiguration and targets the domain of data-centric applications.
- A reactive operational semantics that supports the propagation of changes through a data-flow dependency graph, and allows for reconfiguration of both code and data.
- A monadic type system that statically ensures that both propagation of changes and dynamic reconfiguration happens without breaking type safety.
- Provable type preservation and progress properties, including the convergence of change propagation, to mathematically ensure the soundness of the whole model.

The remainder of this paper is structured as follows. Section 2 introduces our programming language and illustrates the reconfiguration capabilities of our language, by means of two simple examples. Section 3 formally presents the our core reactive and imperative programming language, with its operational semantics and type system. Section 4 states the soundness properties of our language, and also sketches the corresponding proofs. Sections 5 and 6 provide a comparison with related work and provide a final discussion on our work.

2 Reactive Web Programming

Our computational model is designed to host a running instance of a data-centric application, comprising both state and code. We define an incremental and reactive programming environment for data-centric applications, by providing a set of top-level software construction operations. These operations allow the safe definition and deployment of new programming elements, and basic interaction with the application. The application state may be inspected at any time through a set of visible names denoting language values. Intuitively, these visible names represent access points to be linked to elements of a user interface (cf. web pages). Internally, visible names denote either state variables or (values of) expressions on other visible names, representing pure data transformations. We define a monadic model where state variables can be only be updated by explicit execution of (monadic) **action** values at the application's top-level. Pure data transformation names react automatically and implicitly to changes in other visible names. Changes are propagated through the name dependency graph, revisiting all data transformation names that need to be re-evaluated. For the sake of simplicity, we have not optimized the propagation of changes, like for instance [6]. The semantic separation provided by the monadic approach, ensures termination of the propagation of changes.

The language top-level operations comprise the declaration/redefinition of state variables (**var**) and pure data transformation expressions (**def**), and the execution of imperative monadic operations (**do**). For instance, if we apply the following sequence of operations to an empty state

```
var a = 0
def b = a + 1
def c = b + a
do action { a := a + 1 }
```

We obtain access to three visible names `a`, `b`, and `c` denoting values `0`, `1`, and `1`, respectively, before the **do** operation, and `1`, `2`, and `3`, after the **do** operation. On changing state variable `a` with the execution of the **action** value containing the assignment `a := a + 1`, the values denoted by `b` and `c` are automatically updated. Recall that the contents of state variables can only be changed by the execution (**do**) of monadic **action** values. Pure data transformation definitions (**def**), are reactively updated when their defining values change. Notice that initializer expressions in state variables (**var**), and assignment expressions are ignored in the visible names dependency graph, and do not get re-evaluated during propagation of changes. This is an abstract model for data-centric applications, where state variables correspond to the application's persistent data layer, and pure data transformation definitions correspond to the application's logic (e.g. queries). The complementary monadic **action** values contain sequences of delayed assignments whose effect may depend on state. They are used to represent the set of allowed imperative interactions with the application, which can be associated to active elements like buttons or links, and can be seen as a surrogate for user interface events.

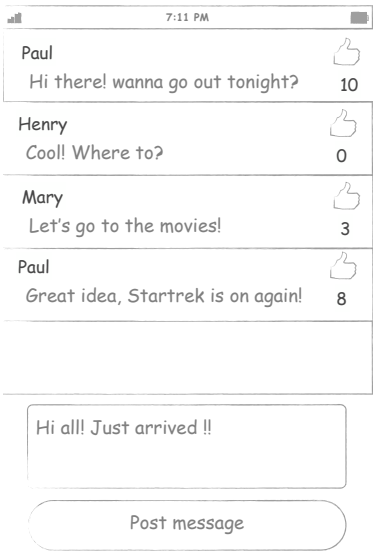

Fig. 1. Bulletin Board User Interface.

Top-level operations are incrementally interpreted to declare/redefine visible names. For instance, the execution of the following operation in the prior state

$$\textbf{def}\ \mathsf{b} = \mathsf{a} + 2$$

redefines name b's expression, thus causing the propagation of its new value, resulting in the denoted values 1, 3, and 4, for names a, b, and c, respectively.

Besides top-level operations, we introduce a second syntactic level that defines a pure functional expression language, a simply typed lambda-calculus conveniently extended with base values and collections. These constructions are adequate to define data-centric applications, where we model database tables as state variables containing collections. We arbitrarily extend the language with extra orthogonal constructs for illustration purposes (e.g. records, conditionals). For the sake of simplicity, assignment expressions are syntactically restricted to the definition expressions of the special monadic **action** values.

Consider the small motivational example of a simple bulletin board application, whose user interface is illustrated in Figure 1. There are several user interface elements that result from querying and transforming the persistent state of the application (the list of messages). Other elements are designed to update the state, like button "Post" and the thumbs-up icons in the list. The code snippet shown in Figure 2 defines this example application in our language. We define the persistent state of the application (state variable messages), some auxiliary definitions (definitions size, new and like), and the visible user interface

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

$$\textbf{var}\ \mathsf{user} = \text{“Henry”}$$

$$\textbf{def}\ \mathsf{size} = \textbf{iter}(\mathsf{messages}, 0, x.y.(y+1))$$

$$\textbf{def}\ \mathsf{new} = \lambda a.\lambda m.\textbf{action}\ \{\ \mathsf{messages} := \mathsf{messages}@[\left\{ \begin{array}{l} \mathit{id} : \mathsf{size}, \\ \mathit{author} : a, \\ \mathit{message} : m, \\ \mathit{likes} : 0 \end{array} \right\}]\ \}$$

$$\textbf{def}\ \mathsf{like} = \lambda i.\textbf{action}\ \{\ \mathsf{messages} :=$$
$$\textbf{iter}(\mathsf{messages}, [\,], x.y.y@[x.\mathit{id} = i\ ?\ \left\{ \begin{array}{l} \mathit{id} : x.\mathit{id}, \\ \mathit{author} : x.\mathit{author}, \\ \mathit{message} : x.\mathit{message}, \\ \mathit{likes} : x.\mathit{likes} + 1 \end{array} \right\} : x])\ \}$$

$$\textbf{def}\ \mathsf{wall} = \textbf{iter}(\mathsf{messages}, [\,], x.y.y@[\left\{ \begin{array}{l} \mathit{message} : x, \\ \mathit{incLikes} : \mathsf{like}\ x.\mathit{id} \end{array} \right\}])$$

$$\textbf{def}\ \mathsf{post} = \lambda m.(\mathsf{new}\ \mathsf{user}\ m)$$

Fig. 2. Bulletin Board Code.

elements (definitions wall and post). Notice that state variable messages elements are records containing the number of “likes” for each message (field *likes*), besides the message *id*, the *author*'s name, and the *message* text. For the sake of simplicity, we store the current logged user in state variable user.

We next add some auxiliary definitions: size is defined by iterating state variable messages and its value is always kept up-to-date with relation to the contents of the state; new is defined to denote an abstraction that returns an **action** value. When executed, this **action** adds a new record to the messages collection, whose values are instantiated by the application of new. Note that although new uses the size name in its **action**, monadic actions contain delayed expressions and do not get re-evaluated. Therefore, when the state variable messages is updated by executing the action in new, it causes a propagation of changes to size. The propagation is not forwarded to the action produced by new since the usage of size is inside the monadic **action**. Similarly, the abstraction like returns an **action** that increments the field *likes* of one element of collection messages. Finally, wall denotes a collection based on messages with an extra (computed) field denoting a closure, which is an **action** (*incLikes*). This collection (wall) corresponds to the list of messages present in the user interface of Figure 1. In wall, each closure *incLikes* corresponds to the active thumbs-up icon on each list element. An event triggered on one of such icons represents a **do** operation on the corresponding **action**, which is linked to that particular message, and has effect on state variable messages when pressed, thus refreshing the list in the user interface. Notice also that an explicit execution of an **action** resulting from

$$\textbf{var}\ \mathsf{whoLikes} = \left\{ \begin{array}{l} \mathit{name} : \text{“Henry”}, \\ \mathit{msgId} : 0 \end{array} \right\} ::[\,]$$

$$\textbf{var}\ \mathsf{msgText} = \textbf{iter}(\mathsf{messages}, [\,], x.y.y@[\left\{ \begin{array}{l} \mathit{id} : x.\mathit{id}, \\ \mathit{author} : x.\mathit{author}, \\ \mathit{message} : x.\mathit{message} \end{array} \right\}])$$

$$\textbf{def}\ \mathsf{new} = \lambda a.\lambda m.\textbf{action}\ \{\ \mathsf{msgText} := \mathsf{msgText}@[\left\{ \begin{array}{l} \mathit{id} : \mathsf{size}, \\ \mathit{author} : a, \\ \mathit{message} : m \end{array} \right\}]\ \}$$

$$\textbf{def}\ \mathsf{like} = \lambda i.\textbf{action}\ \{\ \mathsf{whoLikes} := \mathsf{whoLikes}@[\left\{ \begin{array}{l} \mathit{name} : \mathsf{user}, \\ \mathit{msgId} : i \end{array} \right\}]\ \}$$

$$\textbf{def}\ \mathsf{countLikes} = \lambda i.\textbf{iter}(\mathsf{whoLikes}, 0, x.y.(x.\mathit{msgId} = i\ ?\ (y+1) : y))$$

$$\textbf{def}\ \mathsf{messages} = \textbf{iter}(\mathsf{msgText}, [\,], x.y.y@[\left\{ \begin{array}{l} \mathit{id} : x.\mathit{id}, \\ \mathit{author} : x.\mathit{author}, \\ \mathit{message} : x.\mathit{message}, \\ \mathit{likes} : \mathsf{countLikes}\ x.\mathit{id} \end{array} \right\}])$$

Fig. 3. Bulletin Board Reconfiguration.

abstraction new, updates the value of size, and also the list of messages shown in the interface through name wall. The event triggered by pressing button “Post” (post) in Figure 1 can be simulated in Figure 2 by the top-level operation

$$\textbf{do}\ (\mathsf{post}\ \text{“Hi all! Just arrived !!”})$$

In summary, by adding a new message or clicking a thumbs-up icon on a message, the state variable messages is modified, which in turn causes the propagation of changes to the top-level names that depend on it (e.g. size and wall), causing the application names to react and refresh the user interface.

2.1 Incremental Programming

We now proceed to illustrate how we can keep developing our running example application. A common case in agile methodologies is to start with a simple data model, and gradually evolve it.

Consider that a new requirement was added to store the users that have “liked” messages, instead of simply counting the occurrences. Figure 3 depicts the incremental reconfiguration operations that need to be applied to achieve this. We start by creating a new state variable (whoLikes) in the application state to store the relation between messages and users. We next create a simpler version of message list, based on name messages (*id*, *author*, and *message*), and store it in a new state variable msgText.

Given the reconfigured state, we now redefine the necessary existing top-level names to reach our goal. The insertion of a message (new) is modified to add an element of the new type to msgText instead of messages. The abstraction

$a, b, c, \ldots \in \mathcal{N}$
$x, y, z, \ldots \in \mathcal{V}$

$\mathcal{P}$	::=	$\mathcal{O}_1, \ldots, \mathcal{O}_n$	*(Program)*
$\mathcal{O}$	::=	**def** $a = e$	(Definition)
	|	**var** $a = e$	(Variable)
	|	**do** e	(Do Action)
e	::=	x	(Variable)
	|	$\lambda x{:}\tau.e$	(Abstraction)
	|	$e\ e'$	(Application)
	|	**b**	(Base Values)
	|	a	(Name)
	|	$e\ op\ e'$	(Binary Op.)
	|	$[e_1, \ldots, e_n]$	(Collection)
	|	**iter**$(e, e', x.y.e'')$	(Iterate)
	|	**match** e **with** $x{::}xs \to e' : e''$	(Match)
	|	**action** $\{\ \overline{a := e}\ \}$	(Action)

Fig. 4. Programming Language.

like is redefined is such a way that it updates the whoLikes collection, instead of messages. We declare an auxiliary top-level name countLikes that, for a given message identifier i returns the number of "likes" for that particular message. Finally, the top-level name messages is redefined to point to the new state variables. Notice that the information in messages is the same, but it is now obtained from two state variables msgText and whoLikes (indirectly through countLikes). The reconfiguration happens incrementally, and seamlessly with relation to the elements used in the user interface in Figure 1 (wall and post), that remain unchanged and operating as before.

This simple language enables an incremental and live programming style, achieved by reconfiguration of applications without disrupting the operation and consistency of code and data. Interferences due to data duplication can be questioned in intermediate states of the reconfiguration, but that problem is out of the scope of this paper, and should be addressed in future work. We now describe the formal developments of our programming language.

3 Programming Language

Our programming language, presented in Figure 4, consists in top-level declaration and interaction operations ($\mathcal{O}$), and a monadic λ-calculus as its functional core (e). We assume given an infinite set of names $\mathcal{N}$, and an infinite set of variables $\mathcal{V}$. Top-level operations include the declaration or redefinition of state variables (**var** $a = e$) in the application name space, associated to a top-level name a, with initial value denoted by expression e. Another kind of top-level operation is the declaration or redefinition of pure data transformation expressions

(**def** $a = e$) which associates the top-level name a to the value of expression e with relation to the current state. Functional core expressions in top-level operations may use previously declared top-level names. Finally, the top-level operation **do** e represents the explicit execution of a monadic action denoted by expression e.

In our functional core we include abstraction $\lambda x{:}\tau.e$ and application $e\ e'$ following call-by-value evaluation. For the sake of simplicity we assume the usual sets of base values ($\mathbf{b}$) (i.e. strings, integers, etc.) and corresponding operators (op). Top-level names (a) in expressions are indistinguishably and implicitly coerced to their denotations. To better convey the scenario of data-centric applications we extend the base monadic λ-calculus with collections, with constructor $[e_1, \ldots, e_n]$, and corresponding operations. The iterator $\mathbf{iter}(e, e', x.y.e'')$ denotes the fold-left operation on the collection denoted by expression e, and the iterated expression e''. Variable x denotes the current value 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. The collection destructor $\mathbf{match}\, e\ \mathbf{with}\ x{::}xs \rightarrow e' : e''$ denotes the value of expression e'' in the case of expression e denoting the empty collection, and the value of expression e' in the case of a collection with at least one element x and tail xs. An $\mathbf{action}\,\{\ \overline{a := e}\ \}$ is a monadic value containing a sequence of delayed assignments to state variables. Action values are the only admissible values for expressions in **do** operations.

A program ($\mathcal{P}$) in our language is a sequence of top-level operations that can be issued to a programming environment where a live instance (state and code) resides and evolves.

3.1 Operational Semantics

We define a reactive operational semantics for our programming language by means of two layered small-step reduction relations. We first define the reduction on program configurations $(\mathcal{S}; \mathcal{P}; \mathcal{L})$, where $\mathcal{S}$ is a state mapping top-level names (a) to triples with the form (e, o, s), with e an expression, whose free top-level names range over the domain of $\mathcal{S}$, and o the current denotation for the expression, that can be either undefined ($\square$) or a computed value (v). We write $S[a \mapsto (e, o, s)]$ to denote a state $\mathcal{S}$ where a is associated with (e, o, s). When a top-level name a has a denotational value $\square$ means that a's expression is being evaluated for the first time. Finally, with s a set of top-level names, representing the names that depend on name a and need to be updated when denotation o changes in the state. We call s the subscribers of a.

The language values and denotations are defined by

$$
\begin{array}{rcl}
v & ::= & \mathbf{b} \\
& | & x \\
& | & \lambda x{:}\tau.e \\
& | & [v_1, \ldots, v_n] \\
& | & \mathbf{action}\,\{\ s_1, \ldots, s_n\ \}
\end{array}
$$

that besides the usual base values, variables, λ-abstractions, and collections of values, we also define monadic **action** values. They are values representing state transformations, triggered only at top-level by a **do** operation.

In a program configuration $(\mathcal{S}, \mathcal{P}, \mathcal{L})$, $\mathcal{P}$ represents the sequence of top-level operations to be evaluated, and $\mathcal{L}$ denotes the queue of top-level names that are scheduled to be updated. The reduction relation on program configurations, written $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$, is defined by the rules in Figure 5. It is based on a reduction relation for expression configurations $(\mathcal{S}; e)$, written as $\mathcal{S}; e \longrightarrow e'$, that specifies how expression e reduces to expression e' with relation to state $\mathcal{S}$. It is defined by the rules in Figure 6. To completely understand the semantics we need some auxiliary abbreviations and definitions.

We define $\psi(\mathcal{S}[a \mapsto (e, o, s)], a) = s$ to denote the subscribers of a. If $a \notin \mathsf{dom}(\mathcal{S})$ then $\psi(\mathcal{S}, a) = \emptyset$. The set of free top-level names used in an expression, written $\sigma(e)$, is defined by

$$
\begin{array}{lll}
\sigma(v) & \triangleq \emptyset \quad v \neq \lambda x.e & \sigma([e_1, \ldots, e_n]) \triangleq \sigma(e_1) \cup \ldots \cup \sigma(e_n) \\
\sigma(\lambda x.e) & \triangleq \sigma(e) & \\
\sigma(a) & \triangleq \{a\} & \sigma(\mathbf{iter}(e, e', x.y.e'')) \triangleq \sigma(e) \cup \sigma(e') \cup \sigma(e'') \\
\sigma(e\ e') & \triangleq \sigma(e) \cup \sigma(e') & \\
\sigma(e\ op\ e') & \triangleq \sigma(e) \cup \sigma(e') & \sigma\left(\begin{array}{l}\mathbf{match}\ e\ \mathbf{with} \\ \quad x{::}xs \rightarrow e' : e''\end{array}\right) \triangleq \sigma(e) \cup \sigma(e') \cup \sigma(e'')
\end{array}
$$

Notice that monadic actions are values, and hence the top-level names used by expressions in assignments are ignored in the above definition. This is because changes are not propagated to actions, and assignments are never re-evaluated. Finally, $\mathsf{subscribe}(\mathcal{S}, a, e)$ denotes a state derived from $\mathcal{S}$ where a is added to the subscribers of all names in $\sigma(e)$, and removed from the subscribers of all other names. This is used to keep all subscribers up-to-date.

$$
\begin{array}{rl}
\mathsf{subscribe}(\mathcal{S}, a, e) \triangleq & \{b \mapsto (e', v, s \cup \{a\}) \mid b \in \sigma(e) \wedge \mathcal{S}(b) = (e', v, s)\} \\
& \cup\ \{b \mapsto (e', v, s \setminus \{a\}) \mid b \notin \sigma(e) \wedge \mathcal{S}(b) = (e', v, s)\}
\end{array}
$$

The rules in Figure 5 define the reduction relation for program configurations $(\mathcal{S}, \mathcal{P}, \mathcal{L})$, which is based on the reduction relation on expressions defined by the rules in Figure 6. The reduction is designed to execute an operation in $\mathcal{P}$ at a time, modifying state $\mathcal{S}$ and $\mathcal{L}$ accordingly. The general idea is to use the queue $\mathcal{L}$ to express reactivity, where rules R – DEFINITION, R – VARIABLE, and R – DO ACTION signal that a name must be re-evaluated; and R – COMPUTE is responsible for updating a value in the state. In the end, we signal all subscribers of a to be re-evaluated. Notice that top-level operations only reduce in an empty queue.

Rules R – DEFINITION and R – VARIABLE both add or replace a declared name, in their initial state, and signal that it must be evaluated, by placing it in the queue. In the case of a **def** operation the state is updated to refresh all dependencies of a (see above for the definition of $\mathsf{subscribe}(S, a, e)$), and the new set of subscribers for name a is maintained the same in the final state ($\psi(\mathcal{S}, a)$). In the case of a newly declared name, subscribers will be set to the empty set.

$$\text{(R – DEFINITION)}\quad \frac{\mathcal{S}' = \mathsf{subscribe}(\mathcal{S}, a, e) \qquad s = \psi(\mathcal{S}, a)}{(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])}$$

$$\text{(R – VARIABLE)}\quad \frac{s = \psi(\mathcal{S}, a)}{(\mathcal{S}; \mathbf{var}\ a = e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}; [a])}$$

$$\text{(R – COMPUTE)}\quad \frac{\mathcal{S}(a) = (e, o, s) \qquad \mathcal{S}; e \longrightarrow^* v'}{(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)}$$

$$\text{(R – DO ACTION)}\quad \frac{\mathcal{S}; e \longrightarrow^* \mathbf{action}\ \{\ s_1, \ldots, s_n\ \}}{(\mathcal{S}; \mathbf{do}\ e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}; \mathbf{do}\ \mathbf{action}\ \{\ s_1, \ldots, s_n\ \}, \mathcal{P}; [\,])}$$

$$\text{(R – DO SKIP)}\quad (\mathcal{S}; \mathbf{do}\ \mathbf{action}\ \{\ \cdot\ \}, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}; \mathcal{P}; [\,])$$

$$\text{(R – DO ASSIGN)}\quad \frac{\mathcal{S}(a) = (e'', v, s)}{(\mathcal{S}; \mathbf{do}\ \mathbf{action}\ \{\ a := e, \overline{a' := e'}\ \}, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, v, s)]; \mathbf{do}\ \mathbf{action}\ \{\ \overline{a' := e'}\ \}, \mathcal{P}; [a])}$$

Fig. 5. Program Operational Semantics.

In the case of a **var** operation there is no need to update the state with new subscribers, because imperative operations (initialization and assignments) do not cause propagation of changes.

Rule R – COMPUTE is the only place where values in the state are actually updated. As a result all subscribers (s) of the updated name being computed are placed in the queue for re-evaluation. Notice that R – COMPUTE and R – DO ACTION, depend on a full reduction of the reduction relation on expressions to proceed. So, the convergence result of program reduction depends on termination of the reduction for expressions. Rule R – COMPUTE dequeues a name and evaluates the corresponding expression, updating the state with the result. This propagates the new value through the data dependency graph. By adding the subscribers of a name we are using a data-driven ("push"-reactive) approach, as opposed to a demand-driven (lazy) approach [11].

The top-level **do** operation is reduced only when it encloses an **action** value. Such value is reached by the next reduction rules. Rule R – DO ACTION, reduces expression e in operation **do** e to an **action** value. Rule R – DO ASSIGN executes the first assignment in an action value by overwriting the expression in the program configuration, assigned name a, with the new expression e, and adding the name a to the queue, thus forcing the evaluation of the newly overwritten expression. Finally, R – DO SKIP ignores the empty **action** $\{\ \cdot\ \}$, and continues with the execution of the remaining program $\mathcal{P}$.

Note that, the only reduction rule that does not require an empty queue is R – COMPUTE. This means that we always propagate the pending changes

$$\frac{\mathcal{S}; e \longrightarrow e'}{\mathcal{S}; \mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\ (\text{R – E-CONTEXT}) \qquad \frac{\mathcal{S}(a) = (e, v, s)}{\mathcal{S}; a \longrightarrow v}\ (\text{R – NAME})$$

$$\mathcal{S}; (\lambda x.e)\, v \longrightarrow e\{v/x\}\ (\text{R – APPLICATION}) \qquad \frac{v'' = [\![v\ op\ v']\!]}{\mathcal{S}; v\ op\ v' \longrightarrow^* v''}\ (\text{R – BINARY OP.})$$

$$\mathcal{S}; \mathbf{iter}([\,], v, x.y.e) \longrightarrow v\ (\text{R – ITERATE NIL}) \qquad \mathcal{S}; \mathbf{iter}(v{::}vs, v', x.y.e) \longrightarrow \mathbf{iter}(vs, e\{v/x\}\{v'/y\}, x.y.e)\ (\text{R – ITERATE})$$

$$\mathcal{S}; (\mathbf{match}\,[\,]\ \mathbf{with}\ x{::}xs \to f : g) \longrightarrow g\ (\text{R – MATCH NIL})$$

$$\mathcal{S}; (\mathbf{match}\, v{::}vs\ \mathbf{with}\ x{::}xs \to f : g) \longrightarrow f\{v/x\}\{vs/xs\}\ (\text{R – MATCH})$$

Fig. 6. Expression Operational Semantics.

through the data-flow graph first, until we reach an empty queue. Our type system establishes enough conditions such that we can prove convergence of the update process (see Section 4). In particular, reconfigurations can only happen when no evaluation is taking place. Reduction terminates when reaching a final program configuration of the form $(\mathcal{S}; \cdot; [\,])$.

The reduction relation on expressions, defined by the rules in Figure 6, is based in contexts defined as follows

$$\begin{aligned}
\mathcal{E} \ ::= \ & \mathcal{E}\ op\ e \\
\mid \ & v\ op\ \mathcal{E} \\
\mid \ & \mathcal{E}\ e \\
\mid \ & (\lambda x.e)\, \mathcal{E} \\
\mid \ & [v_1, \ldots, v_i, \mathcal{E}, e_{i+2}, \ldots, e_n] \\
\mid \ & \mathbf{iter}(\mathcal{E}, e, x.y.e') \\
\mid \ & \mathbf{iter}(v, \mathcal{E}, x.y.e) \\
\mid \ & \mathbf{match}\, \mathcal{E}\ \mathbf{with}\ x{::}xs \to e : e'
\end{aligned}$$

and rule R – E-CONTEXT specifies the usual deep expression reduction. Rule (R – NAME) specifies the implicit dereference of a top-level name, reducing it to the corresponding value in state $\mathcal{S}$. R – APPLICATION implements the usual reduction of a call-by-value application. Reduction of binary operations (R – BINARY OP.) is abstracted from the system. The rules dealing with collection iteration (R – ITERATE NIL, R – ITERATE) are specified as expected. Collection destruction (R – MATCH NIL and R – MATCH) is also straight-forward.

3.2 Type System

Our type language is presented in Figure 7, comprising a representative for basic types β, function types $\tau \to \tau'$, types for homogeneous collections (τ^*), and the monadic type `Action` for all action values.

Our type system is divided into a set of interdependent typing judgements to type programs, expressions, and statements. All typing use a common definition

$$
\begin{array}{rcll}
\tau & ::= & \beta & \text{(Basic types)} \\
 & | & \tau \to \tau' & \text{(Function type)} \\
 & | & \tau^* & \text{(Collection type)} \\
 & | & \texttt{Action} & \text{(Action type)}
\end{array}
$$

Fig. 7. Type Language.

of typing environments Γ, which are mappings from variables to types (τ), and names to type annotations. Type annotations are either of the form $\mathbf{def}_\delta(\tau)$ to describe data transformation names, or they are of the form $\mathbf{var}(\tau)$ to describe state variables. The set δ in a type annotation $\mathbf{def}_\delta(\tau)$ gathers all the name dependencies of the expression associated to the annotated name. We do not keep a record of the data dependencies of the state variables' initializer expression.

The typing judgment for programs, written $\Gamma \vdash \mathcal{P}$ asserts that the sequence of top-level operations that composes program $\mathcal{P}$ is well-typed, with relation to the typing environment Γ. In this case, the domain of Γ contains only names. The typing relation on programs is defined by the rules in Figure 8. The typing of top-level operations depends on a second typing judgment, for expressions, written $\Gamma; \delta \vdash e : \tau$ that asserts that expression e has type τ with relation to the typing environment Γ, and conservatively uses, at most, the set of names δ. An invariant of our type system is that $\delta \subseteq \mathsf{dom}(\Gamma)$. Finally, the typing judgment for assignments (statements), written: $\Gamma; \delta \vdash a := e$ asserts that the assignment $a := e$ is well-typed with relation to the typing environment Γ and using at most the set of names δ. This judgment is kept separate for the sake of simplicity and extensibility of the type system with other imperative operations. The rules for expressions and assignments are presented in Figure 9.

The typing rules for programs, in Figure 8, follow the general structure of sequentially examining the top-level operations in a program. A premise to the majority of typing rules regarding name declarations (**def** or **var** operations), is that the declared name cannot be part of a subexpression's dependencies ($a \notin \delta$). This is checked to ensure the absence of circular definitions through the state and data transformations.

In the case of declaration operations, the typing rules are designed to analyze four different cases. The (first) declaration of a name, the redefinition of a name while maintaining the type of the associated expression, the redefinition of a name with type modification, and the replacing of a state variable definition by a data transformation definition. The first case (where $a \notin \mathsf{dom}(\Gamma)$) is covered by rules T – VARIABLE and T – DEFINITION which, after typing subexpression e with dependencies δ, and ensuring the absence of circular dependencies (through condition $a \notin \delta$), proceed with an enriched typing environment to type the remaining program. The second case, where a reconfiguration happens without changing the type of the expression, is covered by rules T – UPDATE VAR E and T – UPDATE DEF E. The interesting case happens in the case of the definition of data

$$\text{(T – VARIABLE)}\quad \frac{\Gamma;\delta \vdash e : \tau \quad a \notin \mathsf{dom}(\Gamma) \quad a \notin \delta \quad \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{P}}{\Gamma \vdash \mathbf{var}\, a = e, \mathcal{P}}$$

$$\text{(T – DEFINITION)}\quad \frac{\Gamma;\delta \vdash e : \tau \quad a \notin \mathsf{dom}(\Gamma) \quad a \notin \delta \quad \Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathcal{P}}{\Gamma \vdash \mathbf{def}\, a = e, \mathcal{P}}$$

$$\text{(T – UPDATE VAR E)}\quad \frac{\Gamma;\delta \vdash e : \tau \quad a \notin \delta \quad \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{P}}{\Gamma, a : \mathbf{var}(\tau) \vdash \mathbf{var}\, a = e, \mathcal{P}}$$

$$\text{(T – UPDATE DEF E)}\quad \frac{\Gamma;\delta' \vdash e : \tau \quad a \notin \delta' \quad \Gamma, a : \mathbf{def}_{\delta'}(\tau) \vdash \mathcal{P}}{\Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathbf{def}\, a = e, \mathcal{P}}$$

$$\text{(T – UPDATE VAR T)}\quad \frac{\Gamma;\delta \vdash e : \tau' \quad a \notin \delta \quad \tau \neq \tau' \quad a \notin \rho(\Gamma) \quad \Gamma, a : \mathbf{var}(\tau') \vdash \mathcal{P}}{\Gamma, a : \mathbf{var}(\tau) \vdash \mathbf{var}\, a = e, \mathcal{P}}$$

$$\text{(T – UPDATE DEF T)}\quad \frac{\Gamma;\delta' \vdash e : \tau' \quad a \notin \delta' \quad \tau \neq \tau' \quad a \notin \rho(\Gamma) \quad \Gamma, a : \mathbf{def}_{\delta'}(\tau') \vdash \mathcal{P}}{\Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathbf{def}\, a = e, \mathcal{P}}$$

$$\text{(T – UPDATE VAR-DEF)}\quad \frac{\Gamma;\delta \vdash e : \tau \quad a \notin \delta \quad \Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathcal{P}}{\Gamma, a : \mathbf{var}(\tau) \vdash \mathbf{def}\, a = e, \mathcal{P}}$$

$$\text{(T – DO)}\quad \frac{\Gamma;\delta \vdash e : \texttt{Action} \quad \Gamma \vdash \mathcal{P}}{\Gamma \vdash \mathbf{do}\, e, \mathcal{P}}$$

Fig. 8. Typing Rules for Programs.

transformation (T – UPDATE DEF E) where the old dependencies (δ) registered in Γ are replaced by the new dependencies (δ') given by the new expression e. This is enough information to continue avoiding circular name dependencies in the following operations. Since the type is maintained, old dependencies are still sound. The third case corresponds to modifying the type of a visible name. In this case we need an extra restriction, which is that no other name definition must depend on the name being redefined, ($a \notin \rho(\Gamma)$). Where $\rho(\Gamma)$ denotes the union of all sets δ in a typing environment Γ.

$$\begin{aligned}
&\rho(\Gamma, a : \mathbf{def}_{\delta}(\tau)) \triangleq \rho(\Gamma) \cup \delta \\
&\rho(\Gamma, a : \mathbf{var}(\tau)) \triangleq \rho(\Gamma) \\
&\rho(\Gamma, x : \tau) \triangleq \rho(\Gamma) \\
&\rho(\cdot) \triangleq \emptyset
\end{aligned}$$

This condition forces that reconfigurations, involving several names and changing types, must be performed in such an order that no broken dependencies ever exist in the application. Finally, the conditions necessary to replace a state variable definition by a data transformation definition, covered by rule T – UPDATE VAR-DEF, are similar to previous cases. However, the remaining program must be typed knowing the new dependencies of the reconfigured name. The dual case is not as simple because, unlike the case above, it would require the re-computing

$$\text{(T – NAME–D)}\quad \Gamma, a : \mathbf{def}_{\delta'}(\tau); \delta \cup \delta' \cup \{a\} \vdash a : \tau \qquad \text{(T – NAME–V)}\quad \Gamma, a : \mathbf{var}(\tau); \delta \cup \{a\} \vdash a : \tau$$

$$\text{(T – VARIABLE)}\quad \Gamma, x : \tau; \delta \vdash x : \tau \qquad \text{(T – BASE VALUE)}\quad \Gamma; \delta \vdash \mathbf{b} : \beta$$

$$\text{(T – ABSTRACTION)}\quad \frac{\Gamma, x : \tau; \delta \vdash e : \tau'}{\Gamma; \delta \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \text{(T – APPLICATION)}\quad \frac{\Gamma; \delta \vdash e : \tau \to \tau' \quad \Gamma; \delta \vdash e' : \tau}{\Gamma; \delta \vdash e\ e' : \tau'}$$

$$\text{(T – BINARY OP.)}\quad \frac{\Gamma; \delta \vdash e : \tau \quad \Gamma; \delta \vdash e' : \tau' \quad op{:}\ \tau \to \tau' \to \tau''}{\Gamma; \delta \vdash e\ op\ e' : \tau''}$$

$$\text{(T – ITERATE)}\quad \frac{\Gamma; \delta \vdash e : \tau^* \quad \Gamma; \delta \vdash e' : \tau' \quad \Gamma, x : \tau, y : \tau'; \delta \vdash e'' : \tau'}{\Gamma; \delta \vdash \mathbf{iter}(e, e', x.y.e'') : \tau'}$$

$$\text{(T – MATCH)}\quad \frac{\Gamma; \delta \vdash e : \tau^* \quad \Gamma, x : \tau, xs : \tau^*; \delta \vdash e' : \tau' \quad \Gamma; \delta \vdash e'' : \tau'}{\Gamma; \delta \vdash \mathbf{match}\, e\, \mathbf{with}\ x{::}xs \to e' : e'' : \tau'} \qquad \text{(T – COLLECTION)}\quad \frac{\Gamma; \delta \vdash e_i : \tau}{\Gamma; \delta \vdash [e_1, \ldots, e_n] : \tau^*}$$

$$\text{(T – ACTION)}\quad \frac{\Gamma; \delta \vdash a_i := e_i}{\Gamma; \delta \vdash \mathbf{action}\, \{\ \overline{a := e}\ \} : \texttt{Action}} \qquad \text{(T – ASSIGN)}\quad \frac{\Gamma, a : \mathbf{var}(\tau); \delta \vdash e : \tau}{\Gamma, a : \mathbf{var}(\tau); \delta \cup \{a\} \vdash a := e}$$

Fig. 9. Typing Rules for Expressions and Assignments.

all dependencies in Γ. In the example of Section 2, we show that several steps are needed to refactor de data layer of an application, always maintaining the type soundness between all the elements.

The typing of a top-level operations of the form **do** e, covered by rule T – DO, requires that expression e is typed with type `Action`. Notice that expression e denotes an **action** value, whose effect is not re-evaluated by change propagation, hence the dependencies of e (δ) are ignored in the bookkeeping process.

The typing rules for expressions, depicted in Figure 9, are quite straightforward. Notice that there is extra information (δ) to store the set of name dependencies of the expression. The most interesting cases are the axiom for names of definitions (T – NAME–D), where dependencies must transitively include the declared dependencies of that particular name, and the axiom for names of state variables (T – NAME–V), that do not depend on other names, and where resulting dependencies must only include the name itself. Notice that δ includes at least the actual real name dependencies of the typed expression. In all other cases, dependencies are simply propagated through the derivation.

4 Type Safety

We next present the formal framework for proving type safety and change propagation convergence, included in a subject reduction and progress theorems.

$$\Gamma \mid \cdot \vdash \cdot$$

$$\frac{\Gamma;\delta \vdash e : \tau \qquad \Gamma;\delta \vdash v : \tau \qquad \Gamma, a : \mathbf{def}_\delta(\tau) \mid \Gamma' \vdash \mathcal{S} \qquad \mathcal{S}, a \mapsto (e, v, s) \vdash [a] \qquad \delta \subseteq \mathsf{dom}(\Gamma) \qquad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{def}_\delta(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, v, s)}$$

$$\frac{\Gamma;\delta \vdash e : \tau \qquad \Gamma, a : \mathbf{def}_\delta(\tau) \mid \Gamma' \vdash \mathcal{S} \qquad \mathcal{S}, a \mapsto (e, \Box, s) \vdash [a] \qquad \delta \subseteq \mathsf{dom}(\Gamma) \qquad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{def}_\delta(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, \Box, s)}$$

$$\frac{\Gamma, a : \mathbf{var}(\tau), \Gamma';\delta \vdash e : \tau \quad \Gamma, a : \mathbf{var}(\tau), \Gamma';\delta' \vdash v : \tau \quad \Gamma, a : \mathbf{var}(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, v, s) \vdash [a] \quad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{var}(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, v, s)}$$

$$\frac{\Gamma;\delta \vdash e : \tau \quad \Gamma, a : \mathbf{var}(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, \Box, s) \vdash [a] \quad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{var}(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, \Box, s)}$$

Fig. 10. Typing Rules for States.

The results presented here follow the standard syntactic approach [21], and are proved by induction on the length of the typing derivations, we provide here a proof sketch when relevant and present the full proofs in Appendix B.

To build the necessary formal background, we start by defining the notion of well-formed queue with relation to a state, written $\mathcal{S} \vdash \mathcal{L}$. It asserts that the subscription relation in state $\mathcal{S}$ establishes a well-founded order on names. This implies that a well-formed queue can reach the empty queue just by following the subscriber relation in state $\mathcal{S}$.

Definition 1 (Well-Formed Queue). *A queue $\mathcal{L}$ is well-formed with relation to a state $\mathcal{S}$, written $\mathcal{S} \vdash \mathcal{L}$, if it can be inductively defined by the rules*

$$\frac{s = \psi(\mathcal{S}, a) \quad \mathcal{S} \vdash \mathcal{L}@s}{\mathcal{S} \vdash a::\mathcal{L}} \qquad \mathcal{S} \vdash [\,]$$

If this property is maintained during the execution we conclude that there are no dependency cycles in the state.

We next define the notion of well-typed state that should also be preserved by the operational semantics. The typing judgment for states, written $\Gamma \mid \Gamma' \vdash \mathcal{S}$ asserts that all names a in a state $\mathcal{S}$ have a corresponding name in typing environment Γ'. The rules that define it are given in Figure 10. The typing environment Γ (disjoint from Γ') describes the names necessary to type the expressions in state $\mathcal{S}$. Notice that $\mathsf{dom}(\mathcal{S})\#\mathsf{dom}(\Gamma)$ and that the expressions in $\mathcal{S}$ are typed in Γ, which again forces a well-founded order on names in the state, by inspection of the data dependencies. The typing of a state implies the typing of all enclosed expressions, and also a well-founded order on names, through name dependency relation contained in typing environment Γ.

The technique of statically imposing an order on names in the typing process resembles the ordered logic approach [15]. We syntactically track the order defined according to the usage of names, i.e. a name only uses names on the left-hand side (hence $\delta \subseteq \mathsf{dom}(\Gamma)$ for **def** cases), and a name is only used by names on its right-hand side (hence $s \subseteq \mathsf{dom}(\Gamma')$ for both **def** and **var** cases). Since the subscribers of a are on the right hand side (Γ') and also in state $\mathcal{S}$, we may state that for the given evaluation state the queue is well-formed, written $\mathcal{S}, a \mapsto (e, o, s) \vdash [a]$.

We can now define a typing relation for program configurations, written $\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})$, if the state $\mathcal{S}$ and program $\mathcal{P}$ are well-typed and the queue $\mathcal{L}$ is well-formed.

$$\frac{\cdot \mid \Gamma \vdash \mathcal{S} \qquad \Gamma \vdash \mathcal{P} \qquad \mathcal{S} \vdash \mathcal{L}}{\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})}$$

Moreover, an expression configuration $(\mathcal{S}; e)$ is well-typed with relation to a typing environment Γ, written $\Gamma \vdash (\mathcal{S}; e)$, if the state $\mathcal{S}$ is well-typed and the expression e is also well-typed.

$$\frac{\cdot \mid \Gamma \vdash \mathcal{S} \qquad \Gamma; \delta \vdash e : \tau}{\Gamma \vdash (\mathcal{S}; e)}$$

Given these premiliminary definitions we can state the soundness properties ensured by our type system and operational semantics. We next state the progress and type preservation properties for expressions.

Lemma 1 (Expressions Progress). *For all expression configurations $(\mathcal{S}; e)$ and typing environments Γ, if $\Gamma \vdash (\mathcal{S}; e)$ and $\Gamma; \delta \vdash e : \tau$ and for all $a \in \delta$ we have $\mathcal{S}(a) = (e'', v, s)$, then either e is a value, or there is an expression e' such that $\mathcal{S}; e \longrightarrow e'$.*

Notice that in order to ensure progress we need to establish that all names potentially used (δ) are defined. Later we keep an invariant of program configurations, that only one name can be undefined at a time ($\square$) and it is not used in e.

Lemma 2 (Expressions Type Preservation). *For all expression configurations $(\mathcal{S}; e)$ and typing environments Γ, if $\Gamma \vdash (\mathcal{S}; e)$ and $\Gamma; \delta \vdash e : \tau$ and an expression reduction $\mathcal{S}; e \longrightarrow e'$, then $\Gamma; \delta \vdash e' : \tau$.*

Lemma 1 and Lemma 2 define type safety for expressions. We next define progress and preservation properties for programs.

Theorem 1 (Progress of Programs). *For all program configurations $(\mathcal{S}; \mathcal{P}; \mathcal{L})$ if $\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})$ and $\forall a \in \mathsf{dom}(\mathcal{S}).\ (\mathcal{S}(a) = (e, \square, s) \Rightarrow \mathcal{L} = [a])$, then there is a program configuration $(\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ such that $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ and $\forall a \in \mathsf{dom}(\mathcal{S}').\ (\mathcal{S}'(a) = (e, \square, s) \Rightarrow \mathcal{L}' = [a])$.*

To ensure the condition of the preservation result for expressions, that no used names can be undefined, we add the extra invariant asserting that the only one name in the state whose denotation can be undefined ($\square$), and that only happens if it is the only element of the queue.

In order to prove the convergence of the evaluation queue, we introduce a measure on names corresponding to the number of all names that must be refreshed on update. In this way we can prove that our measure always decreases during evaluation, and that well-typed programs always reach an empty queue.

Definition 2 (Name Length). *We define the length of a name a with relation to a state $\mathcal{S}$, written $m_{\mathcal{S}}(a)$, as follows:*

$$m_{\mathcal{S}}(a) \triangleq 1 + \sum_{s_i \in \psi(\mathcal{S},a)} m_{\mathcal{S}}(s_i)$$

Definition 3 (Queue Length). *We define the length of a queue $\mathcal{L}$ with relation to a state $\mathcal{S}$, written, $m_{\mathcal{S}}(\mathcal{L})$, as the sum of the lengths of all its names.*

Intuitively, the length of a name represents the number of computation steps needed to propagate an update through the data-flow graph. The length of a queue, represents the computation steps needed to reach an empty queue. We now need to relate the typing relation of program configurations and the definition of the measuring function for a particular queue ($m_{\mathcal{S}}(\mathcal{L})\uparrow$). This result is built around the intermediate results that ensure the absence of dependency cycles in the state. We interchangeably use the measure on names, and the lifted relation for queues without loss of precision.

Lemma 3 (Length Defined). *For all configurations $(\mathcal{S};\mathcal{P};\mathcal{L})$, and typing environments Γ, if $\Gamma \vdash (\mathcal{S},\mathcal{P},\mathcal{L})$ then $m_{\mathcal{S}}(\mathcal{L})\uparrow$.*

Proof. By induction and following the ordering of names provided by the typing relation on states.

Lemma 4 (Preservation & Convergence). *For all configurations $(\mathcal{S};\mathcal{P};\mathcal{L})$ and $(\mathcal{S}';\mathcal{P}';\mathcal{L}')$, and typing environments Γ, if $\Gamma \vdash (\mathcal{S},\mathcal{P},\mathcal{L})$ and $(\mathcal{S};\mathcal{P};\mathcal{L}) \longrightarrow (\mathcal{S}';\mathcal{P}';\mathcal{L}')$ then, there is a typing environment Γ', such that:*

i. $\Gamma' \vdash (\mathcal{S}',\mathcal{P}',\mathcal{L}')$, *and*
ii. if $\mathcal{L} \neq [\,]$ *then* $m_{\mathcal{S}}(\mathcal{L}') < m_{\mathcal{S}}(\mathcal{L})$.

Proof. By induction on the program reduction $(\mathcal{S};\mathcal{P};\mathcal{L}) \longrightarrow (\mathcal{S}';\mathcal{P}';\mathcal{L}')$. We prove for all possible program reductions, with most cases using the definitions of well-typed program configuration and state. The most involved cases are for R – DEFINITION, R – VARIABLE, which involve an inversion property in the typing of states, and R – COMPUTE, where we need to consider the cases of top-level definitions and variables, and carefuly reconstruct well-typed states and queues. (See Appendix B for the full proof.)

Theorem 2 (Programs Type Preservation). *For all configurations $(\mathcal{S};\mathcal{P};\mathcal{L})$ and $(\mathcal{S}';\mathcal{P}';\mathcal{L}')$, and typing environments Γ, if $\Gamma \vdash (\mathcal{S},\mathcal{P},\mathcal{L})$ and $(\mathcal{S};\mathcal{P};\mathcal{L}) \longrightarrow (\mathcal{S}';\mathcal{P}';\mathcal{L}')$ then, there is a typing environment Γ', such that $\Gamma' \vdash (\mathcal{S}',\mathcal{P}',\mathcal{L}')$.*

Proof. Follows directly from case 1 of Lemma 4.

Theorem 3 (Queue Convergence). *For all configurations* $(\mathcal{S};\mathcal{P};\mathcal{L})$ *and* $(\mathcal{S}';\mathcal{P}';\mathcal{L}')$*, and typing environments* Γ*, if* $\Gamma \vdash (\mathcal{S},\mathcal{P},\mathcal{L})$ *then* $(\mathcal{S};\mathcal{P};\mathcal{L}) \longrightarrow^* (\mathcal{S}';\mathcal{P}';[\,])$.

Proof. Follows directly from case 2 of Lemma 4 and Theorem 1 with the decreasing measure $m_{\mathcal{S}}(\mathcal{L})$.

In summary, the main results we extract from the operational semantics and type system are type safety of programs (Theorem 1 and Theorem 2), and convergence of the update process (Theorem 3).

5 Related Work

Data-flow programming languages have been extensively studied in the past, many of them are summarized in a survey [11]. The pure data-flow model considers that every node is pure and without side effects. Likewise, in our language, a node defined by a top-level operation (**def** or **var**) does not have side effects. Many data-flow languages explore parallel computation by computing several non-interfering nodes simultaneously. Although we do not explore this in the present approach, we believe that it is possible and interesting to extend our operational semantics with a dependency analysis between top-level names to allow parallel propagation of changes.

Adaptive Functional Programming (AFP) [1] uses an underlying dependency graph and a change propagation algorithm to adapt the output when input changes. Based on the dependency graph, the algorithm builds a priority queue of nodes, and executes until the queue is emptied. AFP encodes this algorithm into traces in the operational semantics. We use a similar technical approach, wher propagation is also encoded into operational semantics by the queue, and R – COMPUTE reduction rule. A reduction step with R – COMPUTE takes a name from the queue, computes its new value and add its subscribers to the queue, which corresponds to a trace in AFP. Our approach goes further by allowing the redefinition of top-level names, which dynamically changes the dependency graph, hence dependencies between top-level names may change upon reconfiguration.

Reactive systems [8,13,14,16] maintain an ongoing interaction with their environment just like web applications maintain an interaction with users and data repositories. These reactive systems are based on the notions of instants (when a system reacts) and activations (what causes a reaction). In our language, activations correspond to the execution of monadic actions, that cause the system to react. By joining the data-flow properties of our language together with the dependency between top-level names, we are able to encode reactivity. This reactivity is implicitly abstracted by the evaluation queue, i.e. when the queue is not empty the system is reacting, until it reaches an empty queue and the system waits for a new activation. Imperative Reactive languages [3,4,9] combine the

reactive setting with the imperative paradigm. Although, these works combine reactivity with main-stream imperative programming languages, which demands a high degree of expertise to develop reactive programs. Usually, these extensions require programs to be developed with special and explicit constructs to enable reactivity. Functional Reactive Programming is also an area where several approaches have been proposed [13,16,19,20]. However, neither of these approaches combine reactivity with a verified dynamic reconfiguration mechanism as the one presented in our language.

Several works on dynamic reconfiguration and incremental computation of software systems have already been proposed [7,10,18]. These solutions are mostly designed for imperative programming languages. For instance [10] provides dynamic updates in programs by explicitly defining update regions. When the execution of a program reaches an update region, and if a dynamic update is available, then it is applied. In our approach we do not explicitly define update regions, but since they can only occur on top-level names we restrict them to only occur at their execution level, meaning that they cannot occur while a top-level name's value is being computed. In our proposed language we also combine reactivity with dynamic configuration. When a top-level element is redefined, we also update the dependencies between names, meaning that during execution the graph dynamically changes, while maintaining the typing invariants. To the best of our knowledge we are unaware of any work that supports dynamic reconfiguration in reactive and imperative programming languages.

6 Final Remarks

We have presented the syntax, operational semantics and type system for a core reactive and imperative programming language, that supports dynamic reconfiguration of both code and data. Besides supporting dynamic reconfiguration, our operational semantics also supports reactivity by propagating changes that occur in state variables through a dependency graph of top-level names. Meaning that visible names, are always up-to-date with respect to the persistent state of an application. Our type system statically ensures that both data propagation and reconfiguration happens without breaking type safety. We have also stated provable type preservation and progress theorems for programs, including a queue convergence result.

We believe that this is a suitable core model for an agile development tool that assists on the development of data-centric reactive applications. We can expect high productivity gains provided by such a programming environment that, at each construction step and refactoring action, ensures the consistency of code and underlying data.

Our future roadmap includes extensions related to control sharing to support collaborative developments. Other interesting features, related to language pragmatics, involve parallelization of the queue based on separation properties, and the obvious optimizations of avoiding unnecessary computations.

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive Functional Programming. *ACM Transactions on Programming Languages and Systems*, 28(6), 2006.
2. P. Bhattacharya and I. Neamtiu. Dynamic Updates for Web and Cloud Applications. In *Proceedings of APLWACA*, 2010.
3. F. Boussinot. Reactive C: an extension of C to program reactive systems. *Software: Practice and Experience*, 21(4), 1991.
4. F. Boussinot and J.-F. Susini. The SugarCubes tool box: a reactive Java framework. *Software: Practice and Experience*, 28(14), 1998.
5. 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 Programming Language Design and Implementation*, 2013.
6. Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *Proceedings of Programming Language Design and Implementation*, 2012.
7. D. Duggan. Type-based hot swapping of running modules. *Acta Informatica*, 41(4), 2005.
8. N. Halbwachs. Synchronous Programming of Reactive Systems. In *Proceedings of Computer Aided Verification*, 1998.
9. M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of PLDI*, 2009.
10. M. Hicks and S. Nettles. Dynamic Software updating. *ACM Transactions on Programming Languages and Systems*, 27(6), 2005.
11. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1), Mar. 2004.
12. D.-Y. Lin and I. Neamtiu. Collateral Evolution of Applications and Databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009.
13. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of Principles and Practice of Declarative Programming*, 2005.
14. H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, Continued. In *Proceedings of Workshop on Haskell*, 2002.
15. F. Pfenning and R. J. Simmons. Substructural Operational Semantics as Ordered Logic Programming. In *Proceedings of Logic In Computer Science*, 2009.
16. R. R. Pucella. Reactive Programming in Standard ML. In *Proceedings of Conference on Computer Languages*, 1998.
17. G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of Principles of Programming Languages*, 1993.
18. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM TOPLAS*, 29(4), 2007.
19. Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of Programming Language Design and Implementation*, 2000.
20. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Proceedings of International Conference on Functional Programming*, 2001.
21. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.
22. D. Yellin and R. Strom. INC: a language for incremental computations. In *Proceedings of Programming Language Design and Implementation*, 1988.

A Definitions

Definition 1 (Well-Formed Queue). *A queue $\mathcal{L}$ is well-formed with relation to a state $\mathcal{S}$, written $\mathcal{S} \vdash \mathcal{L}$, if it can be inductively defined by the rules*

$$\frac{s = \psi(\mathcal{S}, a) \quad \mathcal{S} \vdash \mathcal{L}@s}{\mathcal{S} \vdash a{::}\mathcal{L}} \qquad \mathcal{S} \vdash [\,]$$

Definition 4 (Well-Typed State). *A state $\mathcal{S}$ is well-typed with relation to two typing environments Γ' and Γ'' with $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Gamma'') = \emptyset$, if $\Gamma' \mid \Gamma'' \vdash \mathcal{S}$ can be inductively defined by the rules*

$$\Gamma \mid \cdot \vdash \cdot$$

$$\frac{\Gamma; \delta \vdash e : \tau \qquad \Gamma; \delta \vdash v : \tau \qquad \Gamma, a : \mathbf{def}_\delta(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, v, s) \vdash [a] \qquad \delta \subseteq \mathsf{dom}(\Gamma) \qquad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{def}_\delta(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, v, s)}$$

$$\frac{\Gamma; \delta \vdash e : \tau \qquad \Gamma, a : \mathbf{def}_\delta(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, \Box, s) \vdash [a] \qquad \delta \subseteq \mathsf{dom}(\Gamma) \qquad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{def}_\delta(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, \Box, s)}$$

$$\frac{\Gamma, a : \mathbf{var}(\tau), \Gamma'; \delta \vdash e : \tau \quad \Gamma, a : \mathbf{var}(\tau), \Gamma'; \delta' \vdash v : \tau \quad \Gamma, a : \mathbf{var}(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, v, s) \vdash [a] \quad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{var}(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, v, s)}$$

$$\frac{\Gamma; \delta \vdash e : \tau \quad \Gamma, a : \mathbf{var}(\tau) \mid \Gamma' \vdash \mathcal{S} \quad \mathcal{S}, a \mapsto (e, \Box, s) \vdash [a] \quad s \subseteq \mathsf{dom}(\Gamma')}{\Gamma \mid a : \mathbf{var}(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, \Box, s)}$$

Definition 5 (Well-Typed Program Configuration). *A program configuration, written $(\mathcal{S}; \mathcal{P}; \mathcal{L})$, is well-typed with relation to a typing environment Γ, if $\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})$ is derivable by the following rule*

$$\frac{\cdot \mid \Gamma \vdash \mathcal{S} \qquad \Gamma \vdash \mathcal{P} \qquad \mathcal{S} \vdash \mathcal{L}}{\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})}$$

Definition 6 (Well-Typed Expression Configuration). *An expression configuration, written $(\mathcal{S}; e)$, is well-typed with relation to a typing environment Γ, if $\Gamma \vdash (\mathcal{S}; e)$ is derivable by the following rule*

$$\frac{\cdot \mid \Gamma \vdash \mathcal{S} \qquad \Gamma; \delta \vdash e : \tau}{\Gamma \vdash (\mathcal{S}; e)}$$

Definition 2 (Name Length). *We define the length of a name a with relation to a state $\mathcal{S}$, written $m_{\mathcal{S}}(a)$, as follows:*

$$m_{\mathcal{S}}(a) \triangleq 1 + \sum_{s_i \in \psi(\mathcal{S}, a)} m_{\mathcal{S}}(s_i)$$

Definition 3 (Queue Length). *We define the length of a queue $\mathcal{L}$ with relation to a state $\mathcal{S}$, written, $m_{\mathcal{S}}(\mathcal{L})$, as the sum of the lengths of all its names.*

Definition 7. *We define the substitution relation for expressions inductively, as follows:*

$$
\begin{array}{lll}
x\{v/x\} & \triangleq v & \\
y\{v/x\} & \triangleq y & \\
a\{v/x\} & \triangleq a & \\
(\lambda x.e)\{v/y\} & \triangleq \lambda x.(e\{v/y\}) & x \neq y \\
(\mathbf{action}\,\{\ \overline{a := e}\ \})\{v/x\} & \triangleq \mathbf{action}\,\{\ \overline{a := e\{v/x\}}\ \} & \\
v\{x/v'\} & \triangleq v & v \neq \lambda x.e \wedge v \neq \mathbf{action}\,\{\ \overline{a := e}\ \} \\
(e\ f)\{v/x\} & \triangleq (e\{v/x\})\,(f\{v/x\}) & \\
(e\ op\ f)\{v/x\} & \triangleq (e\{v/x\})\ op\ (f\{v/x\}) & \\
[e_1, \ldots, e_n]\{v/x\} & \triangleq [e_1\{v/x\}, \ldots, e_n\{v/x\}] & \\
(\mathbf{iter}(e, f, x.y.g))\{v/z\} & \triangleq \mathbf{iter}(e\{v/z\}, f\{v/z\}, x.y.g\{v/z\}) & z \neq x \wedge z \neq y \\
\begin{pmatrix}\mathbf{match}\ e\ \mathbf{with}\\ y{::}ys \rightarrow f : g\end{pmatrix}\{v/x\} & \triangleq \begin{pmatrix}\mathbf{match}\ e\{v/x\}\ \mathbf{with}\\ y{::}ys \rightarrow (f\{v/x\}) : (g\{v/x\})\end{pmatrix} & x \neq y
\end{array}
$$

B Proofs

Lemma 5 (Substitution for Expressions). *For all expression configurations $(\mathcal{S}; e)$ and typing environments Γ, if $\Gamma \vdash (\mathcal{S}; e)$ and $\Gamma, x : \tau'; \delta \vdash e : \tau$ and a value v such that $\Gamma; \delta \vdash v : \tau'$, then $\Gamma \vdash (\mathcal{S}; e\{v/x\})$ and $\Gamma; \delta \vdash e\{v/x\} : \tau$.*

Proof. By induction on the size of the typing derivation and analyzing the last case applied.

1. Variable Substitution

(H1) $\Gamma, x : \tau'; \delta \vdash x : \tau$

(H2) $\Gamma; \delta \vdash v : \tau'$

(3) $\tau = \tau'$ from (H1)

$x\{v/x\} = v$ by def. of Substitution

$\Gamma; \delta \vdash v : \tau$ from (H2, 3)

2. Variable $x \neq y$

(H1) $\Gamma, x : \tau'; \delta \vdash y : \tau$

$\Gamma; \delta \vdash v : \tau'$

$y\{v/x\} = y$ by def. of Substitution

$\Gamma; \delta \vdash y : \tau$

3. Top-level Name

(H1) $\Gamma, x : \tau'; \delta \vdash a : \tau$

(H2) $\Gamma; \delta \vdash v : \tau'$

$a\{v/x\} = a$ by def. of Substitution

$\Gamma; \delta \vdash a : \tau$

4. Action

(H1) $\Gamma, x : \tau'; \delta \vdash \mathbf{action}\,\{\ \overline{a := e}\ \} : \texttt{Action}$

(H2) $\Gamma; \delta \vdash v : \tau'$

$(\mathbf{action}\,\{\ \overline{a := e}\ \})\{v/x\} = \mathbf{action}\,\{\ \overline{a := e\{v/x\}}\ \}$ by def. of Substitution

$i = 1, \ldots, n$

(3) $\Gamma, x : \tau'; \delta \vdash a_i := e_i$ by inv. of T – ACTION in (H1)

$\delta = \delta' \cup \{a\}$

(4) $\Gamma, x : \tau'; \delta' \vdash e_i$ by inv. of T – ASSIGN in (3)

(5) $\Gamma; \delta' \vdash e_i\{v/x\}$ by I.H. with (3, H2)

(6) $\Gamma; \delta' \cup \{a\} \vdash a_i := e_i\{v/x\}$ by T – ASSIGN in (5)

$\Gamma; \delta \vdash \mathbf{action}\,\{\ \overline{a := e\{v/x\}}\ \} : \texttt{Action}$ by T – ACTION with (6)

5. λ-Abstraction

(H1) $\Gamma, x : \tau'; \delta \vdash \lambda y.e : \tau_y \to \tau_e$

(H2) $\Gamma; \delta \vdash v : \tau'$

$x \neq y$

$(\lambda y.e)\{v/x\} = \lambda y.(e\{v/x\})$ by def. of Substitution

(3) $\Gamma, x : \tau', y : \tau_y; \delta \vdash e : \tau_e$ by inv. of T – ABSTRACTION in (H1)

(4) $\Gamma, y : \tau_y; \delta \vdash e\{v/x\} : \tau_e$ by I.H. with (3, H2)

$\Gamma; \delta \vdash \lambda y.(e\{v/x\}) : \tau_e$ by T – ABSTRACTION in (4)

6. Value

(H1) $\Gamma, x : \tau'; \delta \vdash v' : \tau$

(H2) $\Gamma; \delta \vdash v : \tau'$

$v'\{v/x\} = v'$ by def. of Substitution

$\Gamma; \delta \vdash v' : \tau$

7. Application

(H1) $\Gamma, x : \tau'; \delta \vdash e\,e' : \tau$

(H2) $\Gamma; \delta \vdash v : \tau'$

$(e\,e')\{v/x\} = (e\{v/x\})\,(e'\{v/x\})$ by def. of Substitution

(3) $\Gamma, x : \tau'; \delta \vdash e : \tau_a \to \tau$ by inv. of T – APPLICATION on (H1)

(4) $\Gamma, x : \tau'; \delta \vdash e' : \tau_a$ by inv. of T – APPLICATION on (H1)

(5) $\Gamma; \delta \vdash e\{v/x\} : \tau_a \to \tau$ by I.H. with (H2, 3)

(6) $\Gamma; \delta \vdash e'\{v/x\} : \tau_a$ by I.H. with (H2, 4)

$\Gamma; \delta \vdash (e\{v/x\})\,(e'\{v/x\}) : \tau$ by T – APPLICATION with (5, 6)

8. Binary Operation

(H1) $\Gamma, x : \tau'''; \delta \vdash e\ op\ e' : \tau''$

(H2) $\Gamma;\delta \vdash v : \tau'''$
$(e\ op\ e')\{v/x\} = (e\{v/x\})\ op\ (e'\{v/x\})$ by def. of Substitution
$op\colon\ \tau \to \tau' \to \tau''$
(3) $\Gamma, x : \tau''';\delta \vdash e : \tau$ by inv. of T – BINARY OP. on (H1)
(4) $\Gamma, x : \tau''';\delta \vdash e' : \tau'$ by inv. of T – BINARY OP. on (H1)
(5) $\Gamma;\delta \vdash e\{v/x\} : \tau$ by I.H. with (H2, 3)
(6) $\Gamma;\delta \vdash e'\{v/x\} : \tau$ by I.H. with (H2, 4)
$\Gamma;\delta \vdash e\{v/x\}\ op\ e'\{v/x\} : \tau''$ by T – BINARY OP. with (5, 6)

9. Collection

(H1) $\Gamma, x : \tau';\delta \vdash [e_1, \ldots, e_n] : \tau^*$
(H2) $\Gamma;\delta \vdash v : \tau'$
$[e_1, \ldots, e_n]\{v/x\} = [e_1\{v/x\}, \ldots, e_n\{v/x\}]$ by def. of Substitution
(3) $\Gamma, x : \tau';\delta \vdash e_i : \tau$ by inv. of T – COLLECTION on (H1)
$i = 1, \ldots, n$
(4) $\Gamma;\delta \vdash e_i\{v/x\} : \tau$ by I.H. with (H2, 3)
$\Gamma;\delta \vdash [e_1\{v/x\}, \ldots, e_n\{v/x\}] : \tau^*$ by T – COLLECTION with (4)

10. Iterator

(H1) $\Gamma, z : \tau';\delta \vdash \mathbf{iter}(e, e', x.y.e'') : \tau$
(H2) $\Gamma;\delta \vdash v : \tau'$
$\mathbf{iter}(e, e', x.y.e'')\{v/z\} = \mathbf{iter}(e\{v/z\}, e'\{v/z\}, x.y.e''\{v/z\})$
by def. of Substitution
(3) $\Gamma, z : \tau';\delta \vdash e : \tau''^*$ by inv. of T – ITERATE in (H1)
(4) $\Gamma, z : \tau';\delta \vdash e' : \tau$ by inv. of T – ITERATE in (H1)
(5) $\Gamma, z : \tau', x : \tau'', y : \tau;\delta \vdash e'' : \tau$ by inv. of T – ITERATE in (H1)
(6) $\Gamma;\delta \vdash e\{v/z\} : \tau''^*$ by I.H. with (3, H2)
(7) $\Gamma;\delta \vdash e'\{v/z\} : \tau$ by I.H. with (4, H2)
(8) $\Gamma, x : \tau'', y : \tau;\delta \vdash e''\{v/z\} : \tau$ by I.H. with (5, H2)
$\Gamma;\delta \vdash \mathbf{iter}(e\{v/z\}, e'\{v/z\}, x.y.e''\{v/z\}) : \tau$
by T – ITERATE with (6), (7, 8)

11. Match

(H1) $\Gamma, x : \tau';\delta \vdash \mathbf{match}\, e\ \mathbf{with}\ y{::}ys \to e' : e'' : \tau$
(H2) $\Gamma;\delta \vdash v : \tau'$
$(\mathbf{match}\, e\ \mathbf{with}\ y{::}ys \to e' : e''\{v/x\} =$
$\mathbf{match}\, e\{v/x\}\ \mathbf{with}\ y{::}ys \to e'\{v/x\} : e''\{v/x\}$ by def. of Substitution
(3) $\Gamma, x : \tau';\delta \vdash e : \tau''^*$ by inv. of T – MATCH on (H1)
(4) $\Gamma, x : \tau', y : \tau'', ys : \tau''^*;\delta \vdash e' : \tau$ by inv. of T – MATCH on (H1)
(5) $\Gamma, x : \tau';\delta \vdash e'' : \tau$ by inv. of T – MATCH on (H1)
(6) $\Gamma;\delta \vdash e\{v/x\} : \tau''^*$ by I.H. with (H2, 3)
(7) $\Gamma, y : \tau'', ys : \tau''^*;\delta \vdash e'\{v/x\} : \tau$ by I.H. with (H2, 4)
(8) $\Gamma;\delta \vdash e''\{v/x\} : \tau$ by I.H. with (H2, 5)
$\Gamma;\delta \vdash \mathbf{match}\,(e\{v/x\})\ \mathbf{with}\ y{::}ys \to (e'\{v/x\}) : (e''\{v/x\}) : \tau$
by T – MATCH with (6), (7, 8)

□

Lemma 1 (Expressions Progress). *For all expression configurations* $(\mathcal{S}; e)$ *and typing environments* Γ*, if* $\Gamma \vdash (\mathcal{S}; e)$ *and* $\Gamma; \delta \vdash e : \tau$ *and for all* $a \in \delta$ *we have* $\mathcal{S}(a) = (e'', v, s)$*, then either* e *is a value, or there is an expression* e' *such that* $\mathcal{S}; e \longrightarrow e'$*.*

Proof. By induction on the length of the derivation $\Gamma; \delta \vdash e : \tau$.

1. T – NAME–D

(H1) $\Gamma, a : \mathbf{def}_{\delta'}(\tau); \delta \cup \delta' \cup \{a\} \vdash a : \tau$
(H2) $\Gamma \vdash (\mathcal{S}; a)$
(H3) $\mathcal{S}(a) = (e, v, s)$
$\mathcal{S}; a \longrightarrow v$ by R – NAME with (H3)

2. T – NAME–V

(H1) $\Gamma, a : \mathbf{var}(\tau); \delta \cup \{a\} \vdash a : \tau$
(H2) $\Gamma \vdash (\mathcal{S}; a)$
(H3) $\mathcal{S}(a) = (e, v, s)$
$\mathcal{S}; a \longrightarrow v$ by R – NAME with (H3)

3. T – VARIABLE

(H1) $\Gamma; \delta \vdash x : \tau$
x is a value

4. T – ABSTRACTION

(H1) $\Gamma; \delta \vdash \lambda x.e : \tau \to \tau'$
$\lambda x.e$ is a value

5. T – APPLICATION

(H1) $\Gamma; \delta \vdash e_1\, e_2 : \tau'$
(H2) $\Gamma \vdash (\mathcal{S}; e_1\, e_2)$
(3) $\Gamma; \delta \vdash e_1 : \tau \to \tau'$ by inv. of T – APPLICATION in (H1)
(4) $\Gamma; \delta \vdash e_2 : \tau$ by inv. of T – APPLICATION in (H1)
(5) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 6 with (H2)
(6) $\Gamma \vdash (\mathcal{S}; e_1)$ by Definition 6 with (5, 3)
$\mathcal{S}; e_1 \longrightarrow e_1' \vee e_1$ is a value by I.H. with (6, 3)

(7) CASE: $\mathcal{S}; e_1 \longrightarrow e_1'$
$\mathcal{S}; e_1\, e_2 \longrightarrow e_1'\, e_2$ by R – E-CONTEXT with (7)

(8) CASE: e_1 is a value
(9) $\Gamma \vdash (\mathcal{S}; e_2)$ by Definition 6 with (5, 4)
$\mathcal{S}; e_2 \longrightarrow e_2' \vee e_2$ is a value by I.H. with (9, 4)

(10) SCASE: $\mathcal{S}; e_2 \longrightarrow e_2'$
$\mathcal{S}; e_1\, e_2 \longrightarrow e_1\, e_2'$ by R – E-CONTEXT with (8, 10)

(11) SCASE: e_2 is a value
$e_1 = \lambda x.e_1'$ from (3, 8)

$\mathcal{S}; (\lambda x.e'_1)\, e_2 \longrightarrow e'_1\{e_2/x\}$ by R – APPLICATION with (8, 11)

6. T – BINARY OP.

(H1) $\Gamma; \delta \vdash e_1\ op\ e_2 : \texttt{Int}$

(H2) $\Gamma \vdash (\mathcal{S}; e_1 + e_2)$

$op\colon\ \tau \to \tau' \to \tau''$

(3) $\Gamma; \delta \vdash e_1 : \tau$ by inv. of T – BINARY OP. in (H1)

(4) $\Gamma; \delta \vdash e_2 : \tau'$ by inv. of T – BINARY OP. in (H1)

(5) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 6 with (H2)

(6) $\Gamma \vdash (\mathcal{S}; e_1)$ by Definition 6 with (5, 3)

$\mathcal{S}; e_1 \longrightarrow e'_1 \vee e_1$ is a value by I.H. with (6, 3)

(7) CASE: $\mathcal{S}; e_1 \longrightarrow e'_1$

$\mathcal{S}; e_1\ op\ e_2 \longrightarrow e'_1\ op\ e_2$ by R – E-CONTEXT with (7)

(8) CASE: e_1 is a value

(9) $\Gamma \vdash (\mathcal{S}; e_2)$ by Definition 6 with (5, 4)

$\mathcal{S}; e_2 \longrightarrow e'_2 \vee e_2$ is a value by I.H. with (9, 4)

(10) SCASE: $\mathcal{S}; e_2 \longrightarrow e'_2$

$\mathcal{S}; e_1\ op\ e_2 \longrightarrow e_1\ op\ e'_2$ by R – E-CONTEXT with (8, 10)

(11) SCASE: e_2 is a value

$\mathcal{S}; e_1\ op\ e_2 \longrightarrow v'$ by R – BINARY OP. with (8, 11)

7. T – ITERATE

(H1) $\Gamma; \delta \vdash \mathbf{iter}(e_1, e_2, x.y.e_3) : \tau'$

(H2) $\Gamma \vdash (\mathcal{S}; \mathbf{iter}(e_1, e_2, x.y.e_3))$

(3) $\Gamma; \delta \vdash e_1 : \tau^*$ by inv. of T – ITERATE in (H1)

(4) $\Gamma; \delta \vdash e_2 : \tau'$ by inv. of T – ITERATE in (H1)

(5) $\Gamma, x : \tau, y : \tau'; \delta \vdash e_3 : \tau'$ by inv. of T – ITERATE in (H1)

(6) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 6 with (H2)

(7) $\Gamma \vdash (\mathcal{S}; e_1)$ by Definition 6 with (6, 3)

$\mathcal{S}; e_1 \longrightarrow e'_1 \vee e_1$ is a value by I.H. with (7, 3)

(8) CASE: $\mathcal{S}; e_1 \longrightarrow e'_1$

$\mathcal{S}; \mathbf{iter}(e_1, e_2, x.y.e_3) \longrightarrow \mathbf{iter}(e'_1, e_2, x.y.e_3)$ by R – E-CONTEXT with (8)

(9) CASE: e_1 is a value

(10) $\Gamma \vdash (\mathcal{S}; e_2)$ by Definition 6 with (6, 4)

$\mathcal{S}; e_2 \longrightarrow e'_2 \vee e_2$ is a value by I.H. with (10, 4)

(11) SCASE: $\mathcal{S}; e_2 \longrightarrow e'_2$

$\mathcal{S}; \mathbf{iter}(e_1, e_2, x.y.e_3) \longrightarrow \mathbf{iter}(e_1, e'_2, x.y.e_3)$

by R – E-CONTEXT with (9, 11)

(12) SCASE: e_2 is a value

$e_1 = [\,] \vee e_1 = [v_1, \ldots, v_n]$ from (3, 9)

(13) SSCase: $e_1 = [\,]$
$\mathcal{S}; \mathbf{iter}(e_1, e_2, x.y.e_3) \longrightarrow e_2$ by R – ITERATE NIL with (13, 12)

(14) SSCase: $e_1 = [v_1, \ldots, v_n]$
$\mathcal{S}; \mathbf{iter}(e_1, e_2, x.y.e_3) \longrightarrow \mathbf{iter}([v_2, \ldots, v_n], e_3\{v_1/x\}\{e_2/y\}, x.y.e_3)$
by R – ITERATE with (14, 12)

8. T – MATCH
(H1) $\Gamma; \delta \vdash \mathbf{match}\, e_1\ \mathbf{with}\ x{::}xs \to e_2 : e_3 : \tau'$
(H2) $\Gamma \vdash (\mathcal{S}; \mathbf{match}\, e_1\ \mathbf{with}\ x{::}xs \to e_2 : e_3)$
(3) $\Gamma; \delta \vdash e_1 : \tau^*$ by inv. of T – MATCH in (H1)
(4) $\Gamma, x : \tau, xs : \tau^*; \delta \vdash e_2 : \tau'$ by inv. of T – MATCH in (H1)
(5) $\Gamma; \delta \vdash e_3 : \tau'$ by inv. of T – MATCH in (H1)
(6) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 6 with (H2)
(7) $\Gamma \vdash (\mathcal{S}; e_1)$ by Definition 6 with (6, 3)
$\mathcal{S}; e_1 \longrightarrow e_1' \vee e_1$ is a value by I.H. with (7, 3)

(8) Case: $\mathcal{S}; e_1 \longrightarrow e_1'$
$\mathcal{S}; \mathbf{match}\, e_1\ \mathbf{with}\ x{::}xs \to e_2 : e_3 \longrightarrow \mathbf{match}\, e_1'\ \mathbf{with}\ x{::}xs \to e_2 : e_3$
by R – E-CONTEXT with (8)

(9) Case: e_1 is a value
$e_1 = [\,] \vee e_1 = [v_1, \ldots, v_n]$ from (9, 3)

(10) SCase: $e_1 = [\,]$
$\mathcal{S}; \mathbf{match}\, e_1\ \mathbf{with}\ x{::}xs \to e_2 : e_3 \longrightarrow e_3$
by R – MATCH NIL with (10)

(11) SCase: $e_1 = [v_1, \ldots, v_n]$
$\mathcal{S}; \mathbf{match}\, e_1\ \mathbf{with}\ x{::}xs \to e_2 : e_3 \longrightarrow e_2\{v_1/x\}\{[v_2, \ldots, v_n]/xs\}$
by R – MATCH with (11)

9. T – COLLECTION
(H1) $\Gamma; \delta \vdash [e_1, \ldots, e_n] : \tau^*$
(H2) $\Gamma \vdash (\mathcal{S}; [e_1, \ldots, e_n])$
(3) $\Gamma; \delta \vdash e_i : \tau$ by inv. of T – COLLECTION in (H1)
(4) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 6 with (H2)
(5) $\Gamma \vdash (\mathcal{S}; e_i)$ by Definition 6 with (4, 3)
$\mathcal{S}; e_i \longrightarrow e_i' \vee e_i$ is a value by I.H. with (5, 3)

(6) Case: $\mathcal{S}; e_i \longrightarrow e_i'$
$e_0, \ldots, e_{i-1}$ are values
$\mathcal{S}; [e_1, \ldots, e_i, \ldots, e_n] \longrightarrow [e_1, \ldots, e_i', \ldots, e_n]$ by R – E-CONTEXT with (6)

(7) Case: e_i is a value
$e_0, \ldots, e_n$ are values
$[e_1, \ldots, e_n]$ is a value

10. T – BASE VALUE

(H1) $\Gamma; \delta \vdash \mathbf{b} : \beta$

$\mathbf{b}$ is a value

11. T – ACTION

(H1) $\Gamma; \delta \vdash \mathbf{action}\,\{\ \overline{a := e}\ \} : \texttt{Action}$

$\mathbf{action}\,\{\ \overline{a := e}\ \}$ is a value

□

Lemma 6 (Expressions Context Type Preservation). *For all expression configurations $(\mathcal{S}; \mathcal{E}[e])$ and typing environments Γ, if $\Gamma; \delta \vdash e : \tau'$, and $\Gamma; \delta \vdash e' : \tau'$, then $\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$.*

Proof. By induction on the structure of $\mathcal{E}[e]$.

Lemma 2 (Expressions Type Preservation). *For all expression configurations $(\mathcal{S}; e)$ and typing environments Γ, if $\Gamma \vdash (\mathcal{S}; e)$ and $\Gamma; \delta \vdash e : \tau$ and an expression reduction $\mathcal{S}; e \longrightarrow e'$, then $\Gamma; \delta \vdash e' : \tau$.*

Proof. By induction on the length of the expression reduction $\mathcal{S}; e \longrightarrow e'$.

1. R – E-CONTEXT

(H1) $\Gamma \vdash (\mathcal{S}; \mathcal{E}[e])$

(H2) $\Gamma; \delta \vdash \mathcal{E}[e] : \tau$

(H3) $\mathcal{S}; \mathcal{E}[e] \longrightarrow \mathcal{E}[e']$

(4) $\mathcal{S}; e \longrightarrow e'$ by inv. of R – E-CONTEXT in (H3)

(5) Case: $\mathcal{E}[e] = e\ op\ e''$

op: $\tau \to \tau' \to \tau''$

(6) $\Gamma; \delta \vdash e\ op\ e'' : \tau''$ from (H2, 5)

(7) $\Gamma; \delta \vdash e : \tau$ by inv. of T – BINARY OP. in (6)

(8) $\Gamma; \delta \vdash e' : \tau$ by I.H. with (7, 4)

$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau''$ by Lemma 6 with (H2, 7, 8)

(9) Case: $\mathcal{E}[e] = v\ op\ e$

op: $\tau \to \tau' \to \tau''$

(10) $\Gamma; \delta \vdash v\ op\ e : \tau''$ from (H2, 9)

(11) $\Gamma; \delta \vdash e : \tau'$ by inv. of T – BINARY OP. in (10)

(12) $\Gamma; \delta \vdash e' : \tau'$ by I.H. with (11, 4)

$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau''$ by Lemma 6 with (H2, 11, 12)

(13) Case: $\mathcal{E}[e] = e\ e''$

(14) $\Gamma; \delta \vdash e\ e'' : \tau$ from (H2, 13)

(15) $\Gamma; \delta \vdash e : \tau' \to \tau$ by inv. of T – APPLICATION in (14)

(16) $\Gamma; \delta \vdash e' : \tau' \to \tau$ by I.H. with (15, 4)

$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$ by Lemma 6 with (H2, 15, 16)

(17) Case: $\mathcal{E}[e] = (\lambda x.e'')\, e$
(18) $\Gamma; \delta \vdash (\lambda x.e'')\, e : \tau$ — from (H2, 17)
(19) $\Gamma; \delta \vdash e : \tau'$ — by inv. of T – application in (18)
(20) $\Gamma; \delta \vdash e' : \tau'$ — by I.H. with (19, 4)
$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$ — by Lemma 6 with (H2, 19, 20)

(21) Case: $\mathcal{E}[e] = [v_1, \ldots, v_i, e, e_{i+2}, \ldots, e_n]$
(22) $\Gamma; \delta \vdash [v_1, \ldots, v_i, e, e_{i+2}, \ldots, e_n] : \tau'^*$ — from (H2, 21)
(23) $\Gamma; \delta \vdash e : \tau'$ — by inv. of T – collection in (22)
(24) $\Gamma; \delta \vdash e' : \tau'$ — by I.H. with (23, 4)
$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau'^*$ — by Lemma 6 with (H2, 23, 24)

(25) Case: $\mathcal{E}[e] = \mathbf{iter}(e, e'', x.y.e''')$
(26) $\Gamma; \delta \vdash \mathbf{iter}(e, e'', x.y.e''') : \tau$ — from (H2, 25)
(27) $\Gamma; \delta \vdash e : \tau'^*$ — by inv. of T – iterate in (26)
(28) $\Gamma; \delta \vdash e' : \tau'^*$ — by I.H. with (27, 4)
$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$ — by Lemma 6 with (H2, 27, 28)

(29) Case: $\mathcal{E}[e] = \mathbf{iter}(v, e, x.y.e'')$
(30) $\Gamma; \delta \vdash \mathbf{iter}(v, e, x.y.e'') : \tau$ — from (H2, 29)
(31) $\Gamma; \delta \vdash e : \tau$ — by inv. of T – iterate in (30)
(32) $\Gamma; \delta \vdash e' : \tau$ — by I.H. with (31, 4)
$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$ — by Lemma 6 with (H2, 31, 32)

(33) Case: $\mathcal{E}[e] = \mathbf{match}\, e\, \mathbf{with}\ x{::}xs \to e'' : e'''$
(34) $\Gamma; \delta \vdash \mathbf{match}\, e\, \mathbf{with}\ x{::}xs \to e'' : e''' : \tau$ — from (H2, 33)
(35) $\Gamma; \delta \vdash e : \tau'^*$ — by inv. of T – match in (34)
(36) $\Gamma; \delta \vdash e' : \tau'^*$ — by I.H. with (35, 4)
$\Gamma; \delta \vdash \mathcal{E}[e'] : \tau$ — by Lemma 6 with (H2, 35, 36)

2. R – name
(H1) $\Gamma \vdash (\mathcal{S}; a)$
(H2) $\Gamma; \delta \vdash a : \tau$
(H3) $\mathcal{S}; a \longrightarrow v$
(4) $\mathcal{S}(a) = (e, v, s)$ — by inv. of R – name in (H2)
(5) $\cdot \mid \Gamma \vdash \mathcal{S}$ — by Definition 6 with (H1)
$\Gamma; \delta \vdash v : \tau$ — by Definition 4 with (5, 4, H2)

3. R – application
(H1) $\Gamma \vdash (\mathcal{S}; (\lambda x.e)\, v)$
(H2) $\Gamma; \delta \vdash (\lambda x.e)\, v : \tau'$
(H3) $\mathcal{S}; (\lambda x.e)\, v \longrightarrow e\{v/x\}$
(4) $\Gamma; \delta \vdash \lambda x.e : \tau \to \tau'$ — by inv. of T – application in (H2)
(5) $\Gamma; \delta \vdash v : \tau$ — by inv. of T – application in (H2)
(6) $\Gamma, x : \tau; \delta \vdash e : \tau'$ — by inv. of T – abstraction in (4)
$\Gamma; \delta \vdash e\{v/x\} : \tau'$ — by Lemma 5 with (6, 5)

4. R – BINARY OP.

(H1) $\Gamma \vdash (\mathcal{S}; v\ \ op\ \ v')$
(H2) $\Gamma; \delta \vdash v\ \ op\ \ v' : \tau''$
(H3) $\mathcal{S}; v\ \ op\ \ v' \longrightarrow v''$
(4) $op\colon\ \tau \to \tau' \to \tau''$
$\Gamma; \delta \vdash v : \tau$ by inv. of T – BINARY OP. in (H2)
$\Gamma; \delta \vdash v' : \tau'$ by inv. of T – BINARY OP. in (H2)
(5) $v'' = [\![v\ op\ v']\!]$ by inv. of R – BINARY OP. in (H3)
$\Gamma; \delta \vdash v'' : \tau''$ from (4, 5)

5. R – ITERATE NIL

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{iter}([\,], v, x.y.e))$
(H2) $\Gamma; \delta \vdash \mathbf{iter}([\,], v, x.y.e) : \tau$
(H3) $\mathcal{S}; \mathbf{iter}([\,], v, x.y.e) \longrightarrow v$
$\Gamma; \delta \vdash v : \tau$ by inv. of T – ITERATE in (H2)

6. R – ITERATE

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{iter}(z{::}zs, v, x.y.e))$
(H2) $\mathcal{S}; \mathbf{iter}(z{::}zs, v, x.y.e) \longrightarrow \mathbf{iter}(zs, e\{z/x\}\{v/y\}, x.y.e)$
(H3) $\Gamma; \delta \vdash \mathbf{iter}(z{::}zs, v, x.y.e) : \tau'$
(4) $\Gamma; \delta \vdash z{::}zs : \tau^*$ by inv. of T – ITERATE in (H3)
(5) $\Gamma; \delta \vdash v : \tau'$ by inv. of T – ITERATE in (H3)
(6) $\Gamma, x : \tau, y : \tau'; \delta \vdash e : \tau'$ by inv. of T – ITERATE in (H3)
(7) $\Gamma; \delta \vdash z : \tau$ by inv. of T – BINARY OP. in (4)
(8) $\Gamma; \delta \vdash zs : \tau^*$ by inv. of T – BINARY OP. in (4)
(9) $\Gamma, y : \tau'; \delta \vdash e\{z/x\} : \tau'$ by Lemma 5 with (6, 7)
(10) $\Gamma; \delta \vdash e\{z/x\}\{v/y\} : \tau'$ by Lemma 5 with (9, 5)
$\Gamma; \delta \vdash \mathbf{iter}(zs, e\{z/x\}\{v/y\}, x.y.e) : \tau'$ by T – ITERATE with (8, 10, 6)

7. R – MATCH NIL

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{match}\,[\,]\ \mathbf{with}\ \ x{::}xs \to e : e')$
(H2) $\Gamma; \delta \vdash (\mathbf{match}\,v{::}vs\ \mathbf{with}\ \ f{::}x \to xs : g) : \tau$
(H3) $\mathcal{S}; (\mathbf{match}\,[\,]\ \mathbf{with}\ \ x{::}xs \to e : e') \longrightarrow e'$
$\Gamma; \delta \vdash e' : \tau$ by inv. of T – MATCH in (H2)

8. R – MATCH

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{match}\,v{::}vs\ \mathbf{with}\ \ x{::}xs \to e : e')$
(H2) $\Gamma; \delta \vdash (\mathbf{match}\,v{::}vs\ \mathbf{with}\ \ x{::}xs \to e : e') : \tau'$
(H3) $\mathcal{S}; (\mathbf{match}\,v{::}vs\ \mathbf{with}\ \ x{::}xs \to e : e') \longrightarrow e\{v/x\}\{vs/xs\}$
(4) $\Gamma; \delta \vdash v{::}vs : \tau^*$ by inv. of T – MATCH in (H2)
(5) $\Gamma, x : \tau, xs : \tau^*; \delta \vdash e : \tau$ by inv. of T – MATCH in (H2)
(6) $\Gamma; \delta \vdash v : \tau$ by T – BINARY OP. and (4)
(7) $\Gamma; \delta \vdash vs : \tau^*$ by T – BINARY OP. and (4)
(8) $\Gamma, xs : \tau^*; \delta \vdash e\{v/x\}$ by Lemma 5 with (5, 6)
$\Gamma; \delta \vdash e\{v/x\}\{vs/xs\}$ by Lemma 5 with (8, 7)

□

For a given well-typed state such that $\cdot \mid \Gamma \vdash \mathcal{S}$, and by definition every time that we take a name from $\mathcal{S}$ we swap it from the right to the left until we reach $\Gamma \mid \cdot \vdash \cdot$, we can conclude that the domains of Γ and $\mathcal{S}$ are equal.

Corollary 1 (Equal Domains). *For all typing environments Γ and states $\mathcal{S}$, if $\cdot \mid \Gamma \vdash \mathcal{S}$, then $\mathsf{dom}(\Gamma) = \mathsf{dom}(\mathcal{S})$.*

Lemma 7 (Weakening on Well-typed States). *For all states $\mathcal{S}$ and typing environments Γ such that $a \notin \mathsf{dom}(\Gamma)$ and $\cdot \mid \Gamma \vdash \mathcal{S}$ and $\Gamma; \delta \vdash e : \tau$ and $a \mapsto (e, \square, [\,])$,*

a) If $a \notin \mathsf{dom}(\Gamma)$ and $a : \mathbf{def}_\delta(\tau)$ then, $\cdot \mid \Gamma, a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$.
b) If $a \notin \mathsf{dom}(\Gamma)$ and $a : \mathbf{var}(\tau)$ then, $\cdot \mid \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$.

Proof.

1. Definition Name

(H1) $\cdot \mid \Gamma \vdash \mathcal{S}$
(H2) $\Gamma; \delta \vdash e : \tau$
(H3) $a \mapsto (e, \square, [\,])$
(4) $\Gamma \mid \cdot \vdash \cdot$ by Definition 4 in (H1)
(5) $\Gamma, a : \mathbf{def}_\delta(\tau) \mid \cdot \vdash \cdot$ by Definition 4
(6) $a \mapsto (e, \square, [\,]) \vdash [\,]$ by Definition 1
(7) $a \mapsto (e, \square, [\,]) \vdash [a]$ by Definition 1 with (6)
(8) $\delta \subseteq \mathsf{dom}(\Gamma)$ from (H2)
(9) $[\,] \subseteq \emptyset$
(10) $\Gamma \mid a : \mathbf{def}_\delta(\tau) \vdash a \mapsto (e, \square, [\,])$ by Definition 4 with (H2, 5, 7, 8, 9)
$\cdot \mid \Gamma, a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$ by Definition 4 with (4)

2. Variable Name

(H1) $\cdot \mid \Gamma \vdash \mathcal{S}$
(H2) $\Gamma; \delta \vdash e : \tau$
(H3) $a \mapsto (e, \square, [\,])$
(4) $\Gamma \mid \cdot \vdash \cdot$ by Definition 4 in (H1)
(5) $\Gamma, a : \mathbf{var}(\tau) \mid \cdot \vdash \cdot$ by Definition 4
(6) $a \mapsto (e, \square, [\,]) \vdash [\,]$ by Definition 1
(7) $a \mapsto (e, \square, [\,]) \vdash [a]$ by Definition 1 with (6)
(8) $[\,] \subseteq \emptyset$
(9) $\Gamma \mid a : \mathbf{var}(\tau) \vdash a \mapsto (e, \square, [\,])$ by Definition 4 with (H2, 5, 7, 8)
$\cdot \mid \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$ by Definition 4 with (4)

□

Lemma 8 (Strengthening on Well-typed States). *For all states $\mathcal{S}$ and typing environments Γ,*

a) If $\cdot \mid \Gamma, a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$ then, $\cdot \mid \Gamma \vdash \mathcal{S}$.
b) If $\cdot \mid \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$ then, $\cdot \mid \Gamma \vdash \mathcal{S}$.

Proof.

1. Definition Name

(H1) $\cdot \mid \Gamma, a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$

(2) $\Gamma \mid a : \mathbf{def}_\delta(\tau) \vdash a \mapsto (e, \square, [\,])$ by Definition 4 with (H1)

(3) $\Gamma \mid \cdot \vdash \cdot$ by Definition 4

$\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 4 with (3, 2)

2. Variable Name

(H1) $\cdot \mid \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, [\,])$

(2) $\Gamma \mid a : \mathbf{var}(\tau) \vdash a \mapsto (e, \square, [\,])$ by Definition 4 with (H1)

(3) $\Gamma \mid \cdot \vdash \cdot$ by Definition 4

$\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 4 with (3, 2)

□

Theorem 1 (Progress of Programs). *For all program configurations* $(\mathcal{S}; \mathcal{P}; \mathcal{L})$ *if* $\Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L})$ *and* $\forall a \in \mathsf{dom}(\mathcal{S}).\ (\mathcal{S}(a) = (e, \square, s) \Rightarrow \mathcal{L} = [a])$*, then there is a program configuration* $(\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ *such that* $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ *and* $\forall a \in \mathsf{dom}(\mathcal{S}').\ (\mathcal{S}'(a) = (e, \square, s) \Rightarrow \mathcal{L}' = [a])$.

Proof. By induction on the length of the derivation $\Gamma \vdash \mathcal{P}$ we prove for the case where $\mathcal{L} = [\,]$ and $\mathcal{L} \neq [\,]$ while assuming expression reduction termination.

1. $a{::}\mathcal{L} \neq [\,]$

(H1) $\Gamma \vdash (\mathcal{S}; \mathcal{P}; a{::}\mathcal{L})$

(H2) $\forall a \in \mathsf{dom}(\mathcal{S}).\ (\mathcal{S}(a) = (e, \square, s) \Rightarrow a{::}\mathcal{L} = [a])$

(3) $\Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)

(4) $\mathsf{dom}(\Gamma) = \mathsf{dom}(\mathcal{S})$ by Corollary 1 with (H1)

(5) Case: $a : \mathbf{def}_\delta(\tau)$ and $\mathcal{S}(a) = (e, v', s)$

(6) $\Gamma = \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2$

(7) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$

(8) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s)$ by Definition 4 in (3)

$\Gamma_1; \delta \vdash e : \tau$ by Definition 4 in (8)

(9) $\delta \subseteq \mathsf{dom}(\Gamma_1)$ by Definition 4 in (8)

(10) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(\Gamma)$ from (6)

$\delta \subseteq \mathsf{dom}(\mathcal{S})$ from (9, 10, 4)

(11) $\mathcal{S}; e \longrightarrow^* v$ Assuming expression termination

$(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v, s)]; \mathcal{P}; \mathcal{L}@s)$ by R – COMPUTE with (5, 11)

$\mathcal{S}' = \mathcal{S}_2, a \mapsto (e, v, s), \mathcal{S}_1 = \mathcal{S}[a \mapsto (e, v, s)]$ from (7)

$\forall b \in \mathsf{dom}(\mathcal{S}').\ \mathcal{S}'(b) = (e', v'', s')$ from (H2, 5)

(12) Case: $a : \mathbf{def}_\delta(\tau)$ and $\mathcal{S}(a) = (e, \square, s)$

(13) $\Gamma = \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2$

(14) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, \square, s), \mathcal{S}_1$

(15) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, \square, s)$ by Definition 4 in (3)

$\Gamma_1; \delta \vdash e : \tau$ by Definition 4 in (15)

(16) $\delta \subseteq \mathsf{dom}(\Gamma_1)$ by Definition 4 in (15)

(17) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(\Gamma)$
$\delta \subseteq \mathsf{dom}(\mathcal{S})$ from (16, 17, 4)
(18) $\mathcal{S}; e \longrightarrow^* v$ Assuming expression termination
$(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v, s)]; \mathcal{P}; \mathcal{L}@s)$
by R – COMPUTE with (12, 18)
$a{::}\mathcal{L} = [a]$ from (H2, 12)
(19) $\mathcal{S}' = \mathcal{S}[a \mapsto (e, v, s)]$
$\forall a \in \mathsf{dom}(\mathcal{S}').(\mathcal{S}'(a) = (e', v', s'))$ from (19, 12)

(20) CASE: $a : \mathbf{var}(\tau)$ and $\mathcal{S}(a) = (e, v, s)$
(21) $\Gamma' \mid a : \mathbf{var}(\tau), \Gamma'' \vdash \mathcal{S}', a \mapsto (e, v, s)$ by Definition 4 in (3)
$\Gamma', a : \mathbf{var}(\tau), \Gamma''; \delta \vdash e : \tau$ by Definition 4 in (21)
$\delta \in \mathsf{dom}(\mathcal{S})$
$\forall b \in \mathcal{S}.\ \mathcal{S}(b) = (e', v', s')$ from (H2, 20)
(22) $\mathcal{S}; e \longrightarrow^* v''$ Assuming expression termination
$(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v'', s)]; \mathcal{P}; \mathcal{L}@s)$
by R – COMPUTE with (20, 22)

(23) CASE: $a : \mathbf{var}(\tau)$ and $\mathcal{S}(a) = (e, \square, s)$
(24) $\Gamma' \mid a : \mathbf{var}(\tau), \Gamma'' \vdash \mathcal{S}', a \mapsto (e, \square, s)$ by Definition 4 in (3)
$\mathcal{L} = [\,]$ from (H2, 23)
$\Gamma'; \delta \vdash e : \tau$ by Definition 4 in (24)
(25) $\delta \subseteq \mathsf{dom}(\Gamma')$ by Definition 4 in (24)
(26) $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$
$\delta \subseteq \mathsf{dom}(\mathcal{S})$ from (25, 26, 4)
(27) $\mathcal{S}; e \longrightarrow^* v$ Assuming expression termination
$(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v, s)]; \mathcal{P}; \mathcal{L}@s)$
by R – COMPUTE with (23, 27)
$a{::}\mathcal{L} = [a]$ from (H2, 23)
(28) $\mathcal{S}' = \mathcal{S}[a \mapsto (e, v, s)]$
$\forall a \in \mathsf{dom}(\mathcal{S}').(\mathcal{S}'(a) = (e', v', s'))$ from (28, 23)

2. $\mathcal{L} = [\,]$
(H1) $\Gamma \vdash (\mathcal{S}; \mathcal{P}; [\,])$
(H2) $\forall a \in \mathsf{dom}(\mathcal{S}).\ (\mathcal{S}(a) = (e, \square, s) \Rightarrow \mathcal{L} = [a])$
(3) $\Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)
(4) $\mathsf{dom}(\Gamma) = \mathsf{dom}(\mathcal{S})$ by Corollary 1 with (H1)

CASE: T – DEFINITION
(5) $\mathcal{P} = \mathbf{def}\ a = e, \mathcal{P}'$
(6) $\Gamma \vdash \mathbf{def}\ a = e, \mathcal{P}'$ by Definition 5 in (H1) with (5)
(7) $a \notin \mathsf{dom}(\Gamma)$ by inv. of T – DEFINITION in (6)
(8) $a \notin \mathsf{dom}(\mathcal{S})$ by (7, 4)
(9) $\psi(\mathcal{S}, a) = [\,]$ by def. of ψ with (8)
(10) $\mathcal{S}' = \mathsf{subscribe}(\mathcal{S}, a, e)$

$(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}'[a \mapsto (e, \square, [\,])]; \mathcal{P}'; [a])$
by R – DEFINITION with (10, 9)

CASE: T – VARIABLE

(11) $\mathcal{P} = \mathbf{var}\ a = e, \mathcal{P}'$

(12) $\Gamma \vdash \mathbf{var}\ a = e, \mathcal{P}'$ by Definition 5 in (H1) with (11)

(13) $a \notin \mathsf{dom}(\Gamma)$ by inv. of T – VARIABLE in (12)

(14) $a \notin \mathsf{dom}(\mathcal{S})$ by (13, 4)

(15) $\psi(\mathcal{S}, a) = [\,]$ by def. of ψ and (14)

$(\mathcal{S}; \mathbf{var}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, [\,])]; \mathcal{P}'; [a])$
by R – VARIABLE with (15)

CASE: T – DO

(16) $\mathcal{P} = \mathbf{do}\ e, \mathcal{P}'$

(17) $\Gamma \vdash \mathbf{do}\ e, \mathcal{P}'$ by Definition 5 in (H1) with (16)

$\Gamma; \delta \vdash e : \mathtt{Action}$ by inv. of T – DO in (17)

(18) $\mathcal{S}; e \longrightarrow^* \mathbf{action}\ \{\ \overline{a := e}\ \}$ Assuming expression termination

$(\mathcal{S}; \mathbf{do}\ e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}; \mathbf{do}\ \mathbf{action}\ \{\ \overline{a := e}\ \}, \mathcal{P}'; [\,])$
by R – DO ACTION with (18)

CASE: T – UPDATE VAR E

(19) $\mathcal{P} = \mathbf{var}\ a = e, \mathcal{P}'$

(20) $\Gamma = \Gamma', a : \mathbf{var}(\tau)$

$\Gamma', a : \mathbf{var}(\tau) \vdash \mathbf{var}\ a = e, \mathcal{P}'$ by Definition 5 in (H1) with (19)

(21) $a \in \mathsf{dom}(\Gamma', a : \mathbf{var}(\tau))$

(22) $a \in \mathsf{dom}(\mathcal{S})$ by (21, 4)

(23) $\psi(\mathcal{S}, a) = s$ by def. of ψ and (22)

$(\mathcal{S}; \mathbf{var}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}'; [a])$
by R – VARIABLE with (23)

CASE: T – UPDATE VAR T

(24) $\mathcal{P} = \mathbf{var}\ a = e, \mathcal{P}'$

(25) $\Gamma = \Gamma', a : \mathbf{var}(\tau)$

$\Gamma', a : \mathbf{var}(\tau) \vdash \mathbf{var}\ a = e, \mathcal{P}'$ by Definition 5 in (H1) with (24)

(26) $a \in \mathsf{dom}(\Gamma', a : \mathbf{var}(\tau))$ by inv. of T – UPDATE VAR T in (H1)

(27) $a \in \mathsf{dom}(\mathcal{S})$ by (26, 4)

(28) $\psi(\mathcal{S}, a) = s$ by def. of ψ and (27)

$(\mathcal{S}; \mathbf{var}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}'; [a])$
by R – VARIABLE with (28)

CASE: T – UPDATE DEF E

(29) $\mathcal{P} = \mathbf{def}\ a = e, \mathcal{P}'$

(30) $\Gamma = \Gamma', a : \mathbf{def}_\delta(\tau)$

$\Gamma', a : \mathbf{def}_\delta(\tau) \vdash \mathbf{def}\ a = e, \mathcal{P}'$ by Definition 5 in (H1) with (29)

(31) $a \in \mathsf{dom}(\Gamma', a : \mathbf{def}_\delta(\tau))$

(32) $a \in \mathsf{dom}(\mathcal{S})$ by (31, 4)

(33) $\psi(\mathcal{S}, a) = s$ by def. of ψ and (32)
$(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}'; [a])$
by R – DEFINITION with (33)

CASE: T – UPDATE DEF T
(34) $\mathcal{P} = \mathbf{def}\ a = e, \mathcal{P}'$
(35) $\Gamma = \Gamma', a : \mathbf{def}_\delta(\tau)$
$\Gamma', a : \mathbf{def}_\delta(\tau) \vdash \mathbf{def}\ a = e, \mathcal{P}'$
(36) $a \in \mathsf{dom}(\Gamma', a : \mathbf{def}_\delta(\tau))$ by inv. of T – UPDATE DEF T in (H1)
(37) $a \in \mathsf{dom}(\mathcal{S})$ by (36, 4)
(38) $\psi(\mathcal{S}, a) = s$ by def. of ψ and (37)
$(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}'; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}'; [a])$
by R – DEFINITION with (38)

CASE: T – UPDATE VAR-DEF
(39) $\mathcal{P} = \mathbf{def}\ a = e, \mathcal{P}'$
(40) $\Gamma = \Gamma', a : \mathbf{var}(\tau)$
$\Gamma, a : \mathbf{var}(\tau) \vdash \mathbf{def}\ a = e, \mathcal{P}$
(41) $a \in \mathsf{dom}(\Gamma, a : \mathbf{var}(\tau))$
(42) $a \in \mathsf{dom}(\mathcal{S})$ by (41, 4)
(43) $\psi(\mathcal{S}, a) = s$ by def. of ψ and (42)
$(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$
by R – DEFINITION with (43)

□

Lemma 9. *For all states $\mathcal{S}$ and typing environments Γ and Γ', if $\Gamma \mid \Gamma' \vdash \mathcal{S}$ then $\forall a \in \mathsf{dom}(\mathcal{S}).\ m_{\mathcal{S}}(a)\uparrow$.*

Proof.

1. $\Gamma \mid \cdot \vdash \cdot$
trivially true for empty state

2. $\Gamma \mid a : \mathbf{var}(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, o, s)$
(1) $\Gamma, a : \mathbf{var}(\tau) \mid \Gamma' \vdash \mathcal{S}$ by Definition 4
(2) $s \subseteq \mathsf{dom}(\Gamma')$ by Definition 4
$\forall b \in \mathsf{dom}(\mathcal{S}).\ m_{\mathcal{S}}(b)\uparrow$ by I.H. in (1)
(3) $\mathsf{dom}(\Gamma') = \mathsf{dom}(\mathcal{S})$ from Corollary 1 with (1)
(4) $s \subseteq \mathsf{dom}(\mathcal{S})$ from (2, 3)
(5) $m_{\mathcal{S}}(a) = 1 + \sum_{b \in s} m_{\mathcal{S}}(b)$ by Definition 2
$m_{\mathcal{S}}(a)\uparrow$ from (5, 4)

3. $\Gamma \mid a : \mathbf{def}_\delta(\tau), \Gamma' \vdash \mathcal{S}, a \mapsto (e, o, s)$
(1) $\Gamma, a : \mathbf{def}_\delta(\tau) \mid \Gamma' \vdash \mathcal{S}$ by Definition 4
(2) $s \subseteq \mathsf{dom}(\Gamma')$ by Definition 4
$\forall b \in \mathsf{dom}(\mathcal{S}).\ m_{\mathcal{S}}(b)\uparrow$ by I.H. in (1)

(3) $\mathsf{dom}(\Gamma') = \mathsf{dom}(\mathcal{S})$ — from Corollary 1 with (1)
(4) $s \subseteq \mathsf{dom}(\mathcal{S})$ — from (2, 3)
(5) $m_{\mathcal{S}}(a) = 1 + \sum_{b \in s} m_{\mathcal{S}}(b)$ — by Definition 2
$m_{\mathcal{S}}(a) \uparrow$ — from (5, 4)

□

Lemma 3 (Length Defined). *For all configurations $(\mathcal{S}; \mathcal{P}; \mathcal{L})$, and typing environments Γ, if $\Gamma \vdash (\mathcal{S}, \mathcal{P}, \mathcal{L})$ then $m_{\mathcal{S}}(\mathcal{L}) \uparrow$.*

Proof. Follows from Lemma 9 and the condition $\mathcal{L} \subseteq \mathsf{dom}(\mathcal{S})$.

Lemma 4 (Preservation & Convergence). *For all configurations $(\mathcal{S}; \mathcal{P}; \mathcal{L})$ and $(\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$, and typing environments Γ, if $\Gamma \vdash (\mathcal{S}, \mathcal{P}, \mathcal{L})$ and $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ then, there is a typing environment Γ', such that:*

i. $\Gamma' \vdash (\mathcal{S}', \mathcal{P}', \mathcal{L}')$, *and*
ii. if $\mathcal{L} \neq [\,]$ *then* $m_{\mathcal{S}}(\mathcal{L}') < m_{\mathcal{S}}(\mathcal{L})$.

Proof. By induction on the length of the program reduction $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$.

1. R – DEFINITION

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}; [\,])$
(H2) $(\mathcal{S}; \mathbf{def}\ a = e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$
(3) $\Gamma \vdash \mathbf{def}\ a = e, \mathcal{P}$ — by Definition 5 in (H1)
(4) $\mathsf{dom}(\Gamma) = \mathsf{dom}(\mathcal{S})$ — by Corollary 1 with (H1)
(5) $\mathcal{S} \vdash [\,]$ — by Definition 5 in (H1)
(6) $\cdot \mid \Gamma \vdash \mathcal{S}$ — by Definition 5 in (H1)
(7) $\mathcal{S}' = \mathsf{subscribe}(\mathcal{S}, a, e)$ — by inv. of R – DEFINITION in (H2)
(8) $\cdot \mid \Gamma \vdash \mathcal{S}'$ — from (6, 7)

ii. Trivially true because $\mathcal{L} = [\,]$.

i. Proven by analyzing the two sub-cases: $a \notin \mathsf{dom}(\Gamma)$ and $a \in \mathsf{dom}(\Gamma)$.

(9) CASE: $a \notin \mathsf{dom}(\Gamma)$
(10) $a \notin \mathsf{dom}(\mathcal{S})$ — from (4, 9)
(11) $\mathcal{S}'[a \mapsto (e, \square, s)] = \mathcal{S}', a \mapsto (e, \square, s)$ — from (10, 7)
(12) $a \notin \mathsf{dom}(\mathcal{S}')$ — from (11)
(13) $\Gamma; \delta \vdash e : \tau$ — by inv. of T – DEFINITION in (3)
(14) $\delta \subseteq \mathsf{dom}(\Gamma)$ — from (13)
(15) $\Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathcal{P}$ — by inv. of T – DEFINITION in (3)
(16) $s = \psi(\mathcal{S}, a) = [\,]$ — by inv. of R – DEFINITION in (H2, 10)
(17) $\cdot \mid \Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathcal{S}', a \mapsto (e, \square, s)$ — by Lemma 7 with (8, 13, 11, 16)
(18) $\mathcal{S}', a \mapsto (e, \square, [\,]) \vdash [\,]$ — by Definition 1
(19) $\mathcal{S}', a \mapsto (e, \square, [\,]) \vdash [a]$ — by Definition 1 with (18, 16)
(20) $\cdot \mid \Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash \mathcal{S}', a \mapsto (e, \square, s)$
$\Gamma, a : \mathbf{def}_{\delta}(\tau) \vdash (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}, [a])$
— by Definition 5 with (20, 15, 19, 11)

(21) CASE: $a \in \mathsf{dom}(\Gamma)$
$a \in \mathsf{dom}(\mathcal{S}')$ from (4, 21)
$\mathcal{S}'(a) = (e', o, s)$

(22) SCASE: e and e' have equal types (T – UPDATE DEF E)
$\Gamma = \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2$
$\mathcal{S}' = \mathcal{S}'_2, a \mapsto (e', o, s), \mathcal{S}'_1$
(23) $\Gamma_1, \Gamma_2; \delta' \vdash e : \tau$ by inv. of T – UPDATE DEF E in (3)
(24) $a \notin \delta'$ by inv. of T – UPDATE DEF E in (3)
(25) $\Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{P}$ by inv. of T – UPDATE DEF E in (3)
(26) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}'_2, a \mapsto (e', o, s)$ by Definition 5 in (8)
(27) $\delta' \subseteq \mathsf{dom}(\Gamma_1)$
(28) $\Gamma_1 \mid a : \mathbf{def}_{\delta'}(\tau), \Gamma_2 \vdash \mathcal{S}'_2, a \mapsto (e, \square, s)$ from (26, 27)
(29) $\mathcal{S}'_2[a \mapsto (e, \square, s)] \vdash [a]$ by Definition 4 in (28)
(30) $\cdot \mid \Gamma_1, a : \mathbf{def}_{\delta'}(\tau), \Gamma_2 \vdash \mathcal{S}'[a \mapsto (e, \square, s)]$ by Definition 4 with (28)
(31) $\mathcal{S}'[a \mapsto (e, \square, s)] \vdash [a]$ from (29, 30)
$\Gamma \vdash (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$ by Definition 5 with (30, 25, 31)

(32) SCASE: e and e' have different types (T – UPDATE DEF T)
(33) $\Gamma = \Gamma', a : \mathbf{def}_{\delta'}(\tau')$
(34) $\Gamma'; \delta \vdash e : \tau$ by inv. of T – UPDATE DEF T in (3)
(35) $a \notin \delta$ by inv. of T – UPDATE DEF T in (3)
(36) $a \notin \rho(\Gamma')$ by inv. of T – UPDATE DEF T in (3)
(37) $\mathcal{S}' = \mathcal{S}'', a \mapsto (e', o, s)$
(38) $\Gamma' \mid a : \mathbf{def}_{\delta'}(\tau') \vdash a \mapsto (e', o, s)$ by Definition 4 with (8, 36)
(39) $\delta' \subseteq \mathsf{dom}(\Gamma')$ by Definition 4 with (38)
(40) $s \subseteq \emptyset$ by Definition 4 with (38)
(41) $s = [\,]$ from (40)
(42) $\cdot \mid \Gamma' \vdash \mathcal{S}''$ by Lemma 8 with (8, 33, 37)
(43) $\delta \subseteq \mathsf{dom}(\Gamma')$ from (34)
(44) $a \mapsto (e, w, s) \vdash [a]$ by Definition 1 with (41)
(45) $\cdot \mid \Gamma', a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}'', a \mapsto (e, \square, s)$ by Lemma 7 with (42, 34, 44)
(46) $\mathcal{S}'', a \mapsto (e, \square, s) = \mathcal{S}'[a \mapsto (e, \square, s)]$ from (37)
(47) $\cdot \mid \Gamma', a : \mathbf{def}_\delta(\tau) \vdash \mathcal{S}'[a \mapsto (e, \square, s)]$ from (45, 46)
(48) $\Gamma', a : \mathbf{def}_\delta(\tau) \vdash \mathcal{P}$ by inv. of T – UPDATE DEF T in (3)
(49) $\mathcal{S}'[a \mapsto (e, \square, s)] \vdash [a]$ from (44)
$\Gamma', a : \mathbf{def}_\delta(\tau) \vdash (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$
by Definition 5 with (47, 48, 49)

2. R – VARIABLE
(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{var}\, a = e, \mathcal{P}; [\,])$
(H2) $(\mathcal{S}; \mathbf{var}\, a = e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$
(3) $\Gamma \vdash \mathbf{var}\, a = e, \mathcal{P}$ by Definition 5 in (H1)
(4) $\mathsf{dom}(\Gamma) = \mathsf{dom}(\mathcal{S})$ by Corollary 1 with (H1)
$\mathcal{S} \vdash [\,]$ by Definition 5 in (H1)
(5) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)

ii. Trivially true because $\mathcal{L} = [\,]$.
i. Proven by analyzing the two sub-cases: $a \notin \mathsf{dom}(\Gamma)$ and $a \in \mathsf{dom}(\Gamma)$.

(6) Case: $a \notin \mathsf{dom}(\Gamma)$
$a \notin \mathsf{dom}(\mathcal{S})$ from (4, 6)
(7) $\mathcal{S}'[a \mapsto (e, \square, s)] = \mathcal{S}, a \mapsto (e, \square, s)$ from (6)
(8) $\Gamma; \delta \vdash e : \tau$ by inv. of T – VARIABLE in (3)
$\delta \in \mathsf{dom}(\Gamma)$ from (8)
(9) $\Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{P}$ by inv. of T – VARIABLE in (3)
(10) $s = \psi(\mathcal{S}, a) = [\,]$ by inv. of R – VARIABLE in (3)
(11) $\cdot \mid \Gamma, a : \mathbf{var}(\tau) \vdash \mathcal{S}, a \mapsto (e, \square, s)$ by Lemma 7 with (5, 8, 7, 10)
(12) $\mathcal{S}', a \mapsto (e, \square, [\,]) \vdash [\,]$ by Definition 1
(13) $\mathcal{S}', a \mapsto (e, \square, [\,]) \vdash [a]$ by Definition 1 with (12, 10)
$\Gamma, a : \mathbf{var}(\tau) \vdash (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$ by Definition 5 with (11, 9, 13, 7)

(14) Case: Case $a \in \mathsf{dom}(\Gamma)$
$a \in \mathsf{dom}(\mathcal{S})$ from (4, 14)
$\mathcal{S}(a) = (e', o, s)$

(15) SCase: e and e' have equal types (T – UPDATE VAR E)
$\Gamma = \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2$
$\mathcal{S} = \mathcal{S}'_s, a \mapsto (e', o, s), \mathcal{S}'_1$
(16) $\Gamma_1, \Gamma_2; \delta' \vdash e : \tau$ by inv. of T – UPDATE VAR E in (3)
(17) $a \notin \delta'$ by inv. of T – UPDATE VAR E in (3)
(18) $\Gamma_1, \Gamma_2, a : \mathbf{var}(\tau) \vdash \mathcal{P}$ by inv. of T – UPDATE VAR E in (3)
(19) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}'_2, a \mapsto (e', o, s)$ by Definition 4 in (5)
(20) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}'_2, a \mapsto (e, \square, s)$ from (19)
(21) $\mathcal{S}'_2[a \mapsto (e, \square, s)] \vdash [a]$ by Definition 4 in (20)
(22) $\cdot \mid \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}'[a \mapsto (e, \square, s)]$ by Definition 4 with (20)
(23) $\mathcal{S}[a \mapsto (e, \square, s)] \vdash [a]$ from (21) in (22)
$\Gamma \vdash (\mathcal{S}[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$ by Definition 5 with (22, 18, 23)

(24) SCase: e and e' have different types (T – UPDATE VAR T)
(25) $\Gamma = \Gamma', a : \mathbf{var}(\tau')$
(26) $\Gamma'; \delta \vdash e : \tau$ by inv. of T – UPDATE VAR T in (3)
(27) $a \notin \delta$ by inv. of T – UPDATE VAR T in (3)
(28) $a \notin \rho(\Gamma')$ by inv. of T – UPDATE VAR T in (3)
(29) $\mathcal{S} = \mathcal{S}'', a \mapsto (e', o, s)$
(30) $\Gamma' \mid a : \mathbf{var}(\tau) \vdash a \mapsto (e', o, s)$ by Definition 4 with (5, 28)
(31) $s \subseteq \emptyset$ by Definition 4 with (30)
(32) $s = [\,]$ from (31)
(33) $\cdot \mid \Gamma' \vdash \mathcal{S}''$ by Lemma 8 with (5, 25, 29)
(34) $a \mapsto (e, w, s) \vdash [a]$ by Definition 1 with (32)
(35) $\cdot \mid \Gamma', a : \mathbf{var}(\tau) \vdash \mathcal{S}'', a \mapsto (e, \square, s)$ by Lemma 7 with (33, 26, 34)
(36) $\mathcal{S}'', a \mapsto (e, \square, s) = \mathcal{S}'[a \mapsto (e, \square, s)]$ from (29)
(37) $\cdot \mid \Gamma', a : \mathbf{var}(\tau) \vdash \mathcal{S}'[a \mapsto (e, \square, s)]$ from (35, 36)

(38) $\Gamma', a : \mathbf{var}(\tau) \vdash \mathcal{P}$ by inv. of T – UPDATE DEF T in (3)
(39) $\mathcal{S}'[a \mapsto (e, \square, s)] \vdash [a]$ from (34)
$\Gamma', a : \mathbf{var}(\tau) \vdash (\mathcal{S}'[a \mapsto (e, \square, s)]; \mathcal{P}; [a])$
by Definition 5 with (37, 38, 39)

3. R – COMPUTE
(H1) $\Gamma \vdash (\mathcal{S}; \mathcal{P}; a{::}\mathcal{L})$
(H2) $(\mathcal{S}; \mathcal{P}; a{::}\mathcal{L}) \longrightarrow (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)$
(3) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)
(4) $\Gamma \vdash \mathcal{P}$ by Definition 5 in (H1)
(5) $\mathcal{S} \vdash a{::}\mathcal{L}$ by Definition 5 in (H1)
$\mathcal{S}(a) = (e, o, s)$ by inv. of R – COMPUTE in (H2)
(6) $\mathcal{S}; e \longrightarrow^* v'$ by inv. of R – COMPUTE in (H2)
We consider the cases $o = v \vee o = \square$ and $a : \mathbf{def}_\delta(\tau) \vee a : \mathbf{var}(\tau)$.

(7) CASE: $o = v$ and $a : \mathbf{def}_\delta(\tau)$
$\Gamma = \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2$
(8) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, v, s), \mathcal{S}_1$
(9) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v, s)$ by Definition 4 in (3)
(10) $\Gamma; \delta \vdash e : \tau$ by Definition 4 in (9)
$\Gamma; \delta \vdash v : \tau$ by Definition 4 in (9)
(11) $\Gamma_1, a : \mathbf{def}_\delta(\tau) \mid \Gamma_2 \vdash \mathcal{S}_2$ by Definition 4 in (9)
(12) $\mathcal{S}_2, a \mapsto (e, v, s) \vdash [a]$ by Definition 4 in (9)
(13) $\delta \subseteq \mathsf{dom}(\Gamma_1)$ by Definition 4 in (9)
(14) $s \subseteq \mathsf{dom}(\Gamma_2)$ by Definition 4 in (9)
(15) $\Gamma; \delta \vdash v' : \tau$ by Lemma 2 with (10, 6)
(16) $\mathcal{S}_2, a \mapsto (e, v', s) \vdash [a]$ from (12)
(17) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s)$
by Definition 4 with (10, 15, 11, 16, 13, 14)
(18) $\cdot \mid \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ by Definition 4 with (17)
$\mathcal{S}[a \mapsto (e, v', s)] = \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ from (8)
(19) $\mathcal{S} \vdash \mathcal{L}@s$ by Definition 1 in (5)
$\Gamma \vdash (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)$ by Definition 5 with (18, 4, 19)

(20) CASE: $o = \square$ and $a : \mathbf{def}_\delta(\tau)$
$\Gamma = \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2$
(21) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, \square, s), \mathcal{S}_1$
(22) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, \square, s)$ by Definition 4 in (3)
(23) $\Gamma; \delta \vdash e : \tau$ by Definition 4 in (22)
(24) $\Gamma_1, a : \mathbf{def}_\delta(\tau) \mid \Gamma_2 \vdash \mathcal{S}_2$ by Definition 4 in (22)
(25) $\mathcal{S}_2, a \mapsto (e, \square, s) \vdash [a]$ by Definition 4 in (22)
(26) $\delta \subseteq \mathsf{dom}(\Gamma_1)$ by Definition 4 in (22)
(27) $s \subseteq \mathsf{dom}(\Gamma_2)$ by Definition 4 in (22)
(28) $\Gamma; \delta \vdash v' : \tau$ by Lemma 2 with (23, 6)
(29) $\mathcal{S}_2, a \mapsto (e, v', s) \vdash [a]$ from (25)

(30) $\Gamma_1 \mid a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s)$
by Definition 4 with (23, 28, 24, 29, 26, 27)

(31) $\cdot \mid \Gamma_1, a : \mathbf{def}_\delta(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ by Definition 4 with (30)

$\mathcal{S}[a \mapsto (e, v', s)] = \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ from (21)

(32) $\mathcal{S} \vdash \mathcal{L}@s$ by Definition 1 in (5)

$\Gamma \vdash (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)$ by Definition 5 with (31, 4, 32)

(33) Case: $o = v$ and $a : \mathbf{var}(\tau)$

$\Gamma = \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2$

(34) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, v, s), \mathcal{S}_1$

(35) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v, s)$ by Definition 4 in (3)

(36) $\Gamma; \delta \vdash e : \tau$ by Definition 4 in (35)

$\Gamma; \delta \vdash v : \tau$ by Definition 4 in (35)

(37) $\Gamma_1, a : \mathbf{var}(\tau) \mid \Gamma_2 \vdash \mathcal{S}_2$ by Definition 4 in (35)

(38) $\mathcal{S}_2, a \mapsto (e, v, s) \vdash [a]$ by Definition 4 in (35)

(39) $s \subseteq \mathsf{dom}(\Gamma_2)$ by Definition 4 in (35)

(40) $\Gamma; \delta \vdash v' : \tau$ by Lemma 2 with (36, 6)

(41) $\mathcal{S}_2, a \mapsto (e, v', s) \vdash [a]$ from (38)

(42) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s)$
by Definition 4 with (36, 40, 37, 41, 39)

(43) $\cdot \mid \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ by Definition 4 with (42)

$\mathcal{S}[a \mapsto (e, v', s)] = \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ from (34)

(44) $\mathcal{S} \vdash \mathcal{L}@s$ by Definition 1 in (5)

$\Gamma \vdash (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)$ by Definition 5 with (43, 4, 44)

(45) Case: $o = \square$ and $a : \mathbf{var}(\tau)$

$\Gamma = \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2$

(46) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e, \square, s), \mathcal{S}_1$

(47) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, \square, s)$ by Definition 4 in (3)

(48) $\Gamma; \delta \vdash e : \tau$ by Definition 4 in (47)

(49) $\Gamma_1, a : \mathbf{var}(\tau) \mid \Gamma_2 \vdash \mathcal{S}_2$ by Definition 4 in (47)

(50) $\mathcal{S}_2, a \mapsto (e, \square, s) \vdash [a]$ by Definition 4 in (47)

(51) $s \subseteq \mathsf{dom}(\Gamma_2)$ by Definition 4 in (47)

(52) $\Gamma; \delta \vdash v' : \tau$ by Lemma 2 with (48, 6)

(53) $\mathcal{S}_2, a \mapsto (e, v', s) \vdash [a]$ from (50)

(54) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s)$
by Definition 4 with (48, 52, 49, 53, 51)

(55) $\cdot \mid \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ by Definition 4 with (54)

$\mathcal{S}[a \mapsto (e, v', s)] = \mathcal{S}_2, a \mapsto (e, v', s), \mathcal{S}_1$ from (46)

(56) $\mathcal{S} \vdash \mathcal{L}@s$ by Definition 1 in (5)

$\Gamma \vdash (\mathcal{S}[a \mapsto (e, v', s)]; \mathcal{P}; \mathcal{L}@s)$ by Definition 5 with (55, 4, 56)

4. R – DO ACTION

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{do}\, e, \mathcal{P}; [\,])$

(H2) $(\mathcal{S}; \mathbf{do}\, e, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}; \mathbf{do}\, \mathbf{action}\, \{\ \overline{a := e}\ \}, \mathcal{P}; [\,])$

ii. Trivially true because $\mathcal{L} = [\,]$.

i. Is proved below.

(3) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)
(4) $\Gamma \vdash \mathbf{do}\, e, \mathcal{P}$ by Definition 5 in (H1)
(5) $\mathcal{S} \vdash [\,]$ by Definition 5 in (H1)
(6) $\mathcal{S}; e \longrightarrow^* \mathbf{action}\, \{\, \overline{a := e} \,\}$ by inv of R – DO ACTION in (H1)
(7) $\Gamma; \delta \vdash e : \texttt{Action}$ by inv. of T – DO in (4)
(8) $\Gamma \vdash \mathcal{P}$ by inv. of T – DO in (4)
(9) $\Gamma; \delta \vdash \mathbf{action}\, \{\, \overline{a := e} \,\} : \texttt{Action}$ by Lemma 2 with (7, 6)
(10) $\Gamma \vdash \mathbf{do}\, \mathbf{action}\, \{\, \overline{a := e} \,\}, \mathcal{P}$ by T – DO with (9, 8)
$\Gamma \vdash (\mathcal{S}; \mathbf{do}\, \mathbf{action}\, \{\, \overline{a := e} \,\}, \mathcal{P}; [\,])$ by Definition 5 with (3, 10, 5)

5. R – DO ASSIGN

(H1) $\Gamma \vdash (\mathcal{S}; \mathbf{do}\, \mathbf{action}\, \{\, a := e, \overline{a' := e'} \,\}, \mathcal{P}; [\,])$
(H2) $(\mathcal{S}; \mathbf{do}\, \mathbf{action}\, \{\, a := e, \overline{a' := e'} \,\}, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}[a \mapsto (e, v, s)]; \mathbf{do}\, \mathbf{action}\, \{\, \overline{a' := e'} \,\}, \mathcal{P}; [a])$

ii. Trivially true because $\mathcal{L} = [\,]$.

i. Is proved below.

(3) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)
(4) $\Gamma \vdash \mathbf{action}\, \{\, a := e, \overline{a' := e'} \,\}, \mathcal{P}$ by Definition 5 in (H1)
(5) $\Gamma; \delta \vdash \mathbf{action}\, \{\, a := e, \overline{a' := e'} \,\} : \texttt{Action}$ by inv. of T – DO in (4)
(6) $\Gamma \vdash \mathcal{P}$ by inv. of T – DO in (4)
(7) $\Gamma; \delta' \cup \{a\} \vdash a := e$ by inv. of T – ACTION in (5)
$\delta = \delta' \cup \{a\}$ from (7)
(8) $a : \mathbf{var}(\tau) \in \Gamma$ by T – ASSIGN with (7)
(9) $\Gamma; \delta \vdash a'_i := e'_i \quad i = 1, \ldots, n$ by inv. of T – ACTION in (5)
(10) $\Gamma; \delta \vdash \mathbf{action}\, \{\, \overline{a' := e'} \,\}$ by T – ACTION with (9)
(11) $\Gamma \vdash \mathbf{do}\, \mathbf{action}\, \{\, \overline{a' := e'} \,\}, \mathcal{P}$ by T – DO with (10, 6)
(12) $\mathcal{S}(a) = (e'', v, s)$ by inv. of R – DO ASSIGN in (H2)
$\Gamma = \Gamma_1, a : \mathbf{var}(\tau), \Gamma_2$
(13) $\mathcal{S} = \mathcal{S}_2, a \mapsto (e'', v, s), \mathcal{S}_1$
(14) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e'', v, s)$ by Definition 4 in (3)
(15) $\Gamma_1, a : \mathbf{var}(\tau) \mid \Gamma_2 \vdash \mathcal{S}_2$ by Definition 4 in (14)
(16) $\Gamma_1, a : \mathbf{var}(\tau), \Gamma_2; \delta'' \vdash v : \tau$ by Definition 4 in (14)
(17) $\mathcal{S}_2, a \mapsto (e'', v, s) \vdash [a]$ by Definition 4 in (14)
(18) $s \subseteq \mathsf{dom}(\Gamma_2)$ by Definition 4 in (14)
(19) $\Gamma_1, a : \mathbf{var}(\tau), \Gamma_2; \delta' \vdash e : \tau$ by inv. of T – ASSIGN in (7)
(20) $\mathcal{S}_2 \vdash s$ by Definition 1 in (17)
(21) $\mathcal{S}_2, a \mapsto (e, v, s) \vdash [a]$ by Definition 1 in (20)
(22) $\Gamma_1 \mid a : \mathbf{var}(\tau), \Gamma_2 \vdash \mathcal{S}_2, a \mapsto (e, v, s)$ by Definition 4 with (19, 16, 15, 21, 18)
(23) $\cdot \mid \Gamma \vdash \mathcal{S}[a \mapsto (e, v, s)]$ by Definition 4 in (22)
(24) $\mathcal{S}[a \mapsto (e, v, s)] = \mathcal{S}_2, a \mapsto (e, v, s), \mathcal{S}_1$ from (13)
(25) $\mathcal{S}[a \mapsto (e, v, s)] \vdash [a]$ from (21, 24)
$\Gamma \vdash (\mathcal{S}[a \mapsto (e, v, s)]; \mathbf{do}\, \mathbf{action}\, \{\, \overline{a' := e'} \,\}, \mathcal{P}; [a])$ by Definition 5 with (23, 11, 25)

6. R – DO SKIP

(H1) $\Gamma \vdash (\mathcal{S}; \textbf{do action}\,\{\ \cdot\ \}, \mathcal{P}; [\,])$

(H2) $(\mathcal{S}; \textbf{do action}\,\{\ \cdot\ \}, \mathcal{P}; [\,]) \longrightarrow (\mathcal{S}; \mathcal{P}; [\,])$

ii. Trivially true because $\mathcal{L} = [\,]$.

i. Is proved below.

(3) $\Gamma \vdash \textbf{do action}\,\{\ \cdot\ \}, \mathcal{P}$ by Definition 5 in (H1)

(4) $\Gamma \vdash \mathcal{P}$ by inv. of T – DO in (3)

(5) $\cdot \mid \Gamma \vdash \mathcal{S}$ by Definition 5 in (H1)

(6) $\mathcal{S} \vdash [\,]$ by Definition 5 in (H1)

$\Gamma \vdash (\mathcal{S}; \mathcal{P}; [\,])$ by Definition 1 with (5, 4, 6)

□

Theorem 2 (Programs Type Preservation). *For all configurations* $(\mathcal{S}; \mathcal{P}; \mathcal{L})$ *and* $(\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$*, and typing environments* Γ*, if* $\Gamma \vdash (\mathcal{S}, \mathcal{P}, \mathcal{L})$ *and* $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$ *then, there is a typing environment* Γ'*, such that* $\Gamma' \vdash (\mathcal{S}', \mathcal{P}', \mathcal{L}')$*.*

Proof. Follows directly from case 1 of Lemma 4.

Theorem 3 (Queue Convergence). *For all configurations* $(\mathcal{S}; \mathcal{P}; \mathcal{L})$ *and* $(\mathcal{S}'; \mathcal{P}'; \mathcal{L}')$*, and typing environments* Γ*, if* $\Gamma \vdash (\mathcal{S}, \mathcal{P}, \mathcal{L})$ *then* $(\mathcal{S}; \mathcal{P}; \mathcal{L}) \longrightarrow^* (\mathcal{S}'; \mathcal{P}'; [\,])$*.*

Proof. Follows directly from case 2 of Lemma 4 and Theorem 1 with the decreasing measure $m_{\mathcal{S}}(\mathcal{L})$.