Type-Safe Evolution of Data-Centric Applications

Technical Report (extended version with proofs)

Miguel Domingues  João Costa Seco
CITI – Universidade Nova de Lisboa

Abstract

This paper introduces a uniform, language-based reconfiguration mechanism for data-centric applications, that supports a novel and flexible style of live programming. Our programming model supports the incremental construction and maintenance of applications, by means of verified operations, applied to running instances of software systems. We allow for the seamless, type-safe, evolution and reconfiguration of both application logic and persistent state. Which means that applications may evolve without any service disruption.

We use a data-flow reactive language semantics that safely combines imperative constructs with a reactive context. Our type system ensures the change propagation process through the data dependencies graph, and the reconfiguration of applications are sound. We also prove that type safety implies the convergence of the propagation process, which is essential to correctly implement the reactive behavior of applications.

1. Introduction

Flexible development frameworks and agile development methodologies are gaining in popularity, especially in the domain of web (data-centric) applications. However, the promised incremental and flexible design comes at a price, it is often the case that a large amount of effort is wasted in the redefinition and refactorization of existing code. There are methodologies and development tools that provide mechanisms to soundly perform refactoring applications, but they do not avoid the burden that a code or data-schema update may cause. Updates in running systems usually mean significant service downtime, and often require explicitly programmed scripts to handle the evolution of persistent data. We aim at providing an effective incremental mechanism, that allows safe updates of both application code and data, without causing any sort of service disruption.

We set our programming model in a context where the design of reactive software architectures is increasingly important. A usual requirement for web and cloud applications, as well as in other models [1], is that data changes are automatically transported from the application’s persistent data layer to the (active) user interfaces. This kind of reactive behavior is usually defined by ad-hoc handcrafted code, which is more complex to write and verify, in the presence of distribution and collaborative applications. Our programming language uses suitable language abstractions to capture the reactive nature of data-centric applications, thus allowing a typing verification to reason about its properties.

Our attention is drawn to the problem of how to make the evolution of reactive data-centric applications less time consuming and error-prone. Current development environments are already implementing immediate feedback and preview features, as a crucial factor for productivity. We aim at pushing that frontier even further, by providing a style of incremental [6, 18, 25] and live [5] programming, that captures a core set of construction operations on running systems. Our approach allows developers to continuously evolve an application, by changing its code [8, 11, 20], and reconfiguring the underlying data [2, 13], in a seamless development process. We introduce a reactive programming language, that presents an abstraction level suitable for building data-centric (imperative) applications, where it is possible to establish a dynamic reconfiguration semantics, of both application code and data.

In our language, we have three key ingredients to model our target scenario of data-centric applications: state variables, data transformation expressions, and actions. State variables are used to model the persistent layer of the application, data transformation expressions model the logic that queries data and presents the user with web pages, and actions, that enclose imperative updates to the data layer. In our model, actions are monad-like values that represent delayed imperative computations, and are typically linked to event handlers (UI). The language that we use to model the application logic is a lambda-calculus with collections and records, that we can compositionally change to a more expressive fragment. We follow the model of having a systems’ console to manage, and in our case build, the evolution of a system (cf. [19, 20]). For each of these key ingredients, we define a corresponding top-level operation to be interpreted at the systems’ console. The operations are: the declaration of state variables (\texttt{var}), the declaration of pure data transformations (\texttt{def}), and the explicit execution (\texttt{do}) of actions (\texttt{action}). This model is partially inspired by the usual architectures and conventions of web applications, where the resources of a system (named values) are obtained through GET requests using URLs, and execution of actions correspond to the handling of user actions (cf. POST requests). Changes to the state of the application, caused only by the execution (\texttt{do}) of actions, are implicitly and automatically propagated through the definitions of pure data transformation elements, thus updating their computed values (cf. a data-driven operational semantics [12]). Data dependencies between named elements form a data-flow graph that defines how changes are propagated throughout. This data-flow graph is built

[Copyright notice will appear here once 'preprint' option is removed.]
in such a way, that imperative code (variable initialization and assignments) is kept apart from the change propagation process. In this way, we statically certify the safe use of imperative constructs in a reactive context, by ensuring the absence of infinite change propagation loops.

In summary, we provide a flexible style of live programming, where applications are incrementally built and maintained by sequences of type-safe operations. To the best of our knowledge, this is the first work combining a type-safe reactive and imperative language, with a statically verified dynamic reconfiguration mechanism. Our key contributions can be summarized as follows:

- a novel core reactive language for data-centric applications, that supports type-safe dynamic reconfiguration.
- a data-flow operational semantics supporting the synchronized evolution of both code and data.
- a type system that statically ensures the safety of applications in the presence of both reactive propagation of changes and dynamic reconfiguration operations.
- type preservation and progress results that include the convergence of change propagation.

The remainder of this paper is structured as follows. Section 2 introduces our programming language by means of a small example. Another example, yet more realistic, is presented in Section 3. In Section 4, we formally present our core reactive and imperative programming language, its operational semantics and type system. We then extend our programming language, in Section 5, to support dynamic reconfiguration. Section 6 states the soundness properties of our language, and also sketches the corresponding proofs. Sections 8 and 9 provide a comparison with related work, and a final discussion on our work.

2. Incremental Reactive Programming

We now define our reactive and incremental computational model. This model is designed to host a running instance of a data-centric application, comprising both application logic and persistent state. Applications are incrementally built by a set of top-level construction operations, that enable the safe definition and deployment of named programming elements, and basic interaction with the application persistent state. Existing named programming elements may also be redefined using the same top-level construction operations, thus allowing to dynamically reconfigure the application.

Our programming language comprises two named programming elements: state variables (var) and pure data transformations (def). State variables correspond to the application persistent state, and pure data transformations correspond to the application’s logic. Together, these elements form a data-flow graph based on the dependencies between the usage of names. This dependency graph enable us to propagate changes between elements, i.e., changes occurring at the application persistent state are immediately and seamlessly propagated into pure data transformation elements. By propagating changes, all named elements are always kept up-to-date with respect to the application persistent state. Note that changes are not propagated into state variables (i.e. persistent state). The application persistent state is only changed by the explicit execution (do) of imperative statements (action).

Consider the example from Figure 1. On the left-hand side we present a sequence of operations that declare state variables (var), pure data transformations (def), and execution of actions (do). On the right-hand side we show the corresponding data-flow graph built according to the name dependencies between programming elements. The operations in lines 1 to 3, define names a, b, and c denoting values 0, 1, and 1, respectively. Notice that a is a state variable, and names b and c are pure data transformations. The underlying data-flow graph is incrementally built, along with the application’s programming elements definitions, and kept up-to-date with relation to the name usage of the defined names. In the data-flow graph of Figure 1, nodes correspond to programming elements (either var or def), and edges represent name dependencies, and are directed to denote the propagation of changes. Edges are established, and added to the data-flow graph, based on the definition of each name. For instance, name c is defined by expression a + b, resulting in two edges, one from a to c, and another from b to c. Notice that, in order to ensure the convergence of the change propagation, we must ensure that the data-flow graph is acyclic. This is statically ensured by our type system (presented in Section 4.2), and is one of our final results, expressed in Theorem 3 (Section 6). Notice that the expressiveness of the language is not limited by the acyclic property of the graph, because the application logic can be concentrated in the data transformation expressions, similarly to web application actions.

In line 4, there is an operation do over an action value containing the assignment statement a := a + b. The execution of this action changes the value of the state variable a. Recall that the contents of state variables (var) can only be changed by the execution (do) of action values, which contain sequences of delayed imperative statements. After changing state variable a, the new value will be propagated through the graph, and the values denoted by b and c are reactively updated. When the propagation of changes terminates, names a, b, and c will denote the values 1, 2 and 3, respectively. Pure data transformations (def) are reactively updated when there are changes in the values of the names they depend on. This implies that initializer expressions of pure data transformations (def) are re-evaluated during the propagation of changes. However, initializer expressions of state variables (var) and assignment statements enclosed by action values are ignored during the propagation of changes.

This style of incremental programming enable us to evolve applications in a safe manner. We are able, not only to add new programming elements, but also to redefine the existing ones. Consider the operation in line 5. It redefines the expression associated to name b, thus changing its denotation by evaluating the new expression, and propagating its new value through the dependency graph. After propagating this change, name a remains unchanged with value 1, and names b and c now denote 3 and 4, respectively. Notice that the assignment operation (line 4) is not executed again, even if it uses b. This effect is controlled by the distinction between visible names, and delayed imperative statements enclosed in action values. Notice

3. Example

Consider the small, and yet more realistic, example of a bulletin board application, whose user interface is illustrated in Figure 3. This user interface is built using the following elements: a list of messages with data retrieved from a database; a “thumbs-up” icon for each element in the list, linked to a request that increments a counter of “likes” for each message; a text box that accepts text for
var messages = { id: 0, author: "Paul", message: "Hi there! ...", likes: 10 } :: [...]

def size = iter(messages, 0, x.y.(y + 1))

def new = λa. λm. action { messages := messages@[ id : size, author : a, message : m, likes : 0 ] }

def like = λid. action { messages := [ m.id = id ? [ id : m.id, author : m.author, message : m.message, likes : m.likes + 1 ] : m | m ← messages ] }

def wall = { { message : m, incLikes : like m.id } | m ← messages }

Figure 2. Bulletin Board Code.

Paul
Hi there! wanna go out tonight?

Henry
Cool! Where to?

Mary
Let’s go to the movies!

Paul
Great idea, Starrek is on again!

I’ll pick you up at 6.

Post message Clear filter

Figure 3. Bulletin Board UI Mockup.

a new message; and a button with label “Post message”, linked to a request to add a new message.

We illustrate the sequence of construction operations that may be used to support such a user interface (see Figure 2 for the full program). Starting from an empty system, we first need to define the persistent state of the application. In this case, we need to define a state variable containing a collection of messages (records), with operation

```
var messages = { id : 0, author : "Paul", message : "Hi there! ...", likes : 10 } :: [...]
```

Each element of the collection in state variable messages contains a message identifier (id), the author’s name, the message text, and the number of likes for each message. Name messages is now available to be used in other definitions, for instance, we can write

```
def size = iter(messages, 0, x.y.(y + 1))
```

to compute the size of the collection, which is bound to name size, and is always up-to-date with relation to the contents of state variable messages. The iter expression iterates over the collection using a fold-left semantics. In the iteration body (x.y.(y + 1)), variable x denotes the current element of the collection, and variable y denotes the computed result in the previous iteration, or the initial value (0).

At this stage, these names, can conceivably be accessed through a conventional URL, to be used in a web interface such as the one in Figure 3.

To add a new message, one may use the operation

```
do action { messages := messages@[m] }
```

where m is a message that is added to the collection. We can abstract this operation using a function like

```
def new = λa. λm. let msg = { id : size, author : a, message : m, likes : 0 } in
    action { messages := messages@[msg] }
```

and then link the button “Post message” in the user interface, through a POST request with arguments, to the execution of

```
do (new "Paul" “I’ll pick you up at 6.”)
```

Notice that the definition of new uses name size to assign an identifier to the new message, which is always up-to-date with relation to the collection stored in messages.

In order to continue building the list shown in the user interface, we need to add some active behavior to the “thumbs-up” icon, for each row. So, we need to define a function that increments the counter of a message.

```
def like =
```

```
let inc = λx: { id : x.id, author : x.author, message : x.message, likes : x.likes + 1 } in
λid.action { messages := [ m.id = id ? (inc m) : m | m ← messages ] }
```

2014/6/9
Function `like` takes a message identifier (id) and iterates the collection `messages` using a for-comprehension style [21], and increments the counter of the selected message. We can now use this definition to produce a collection suitable for displaying messages in the user interface, together with the intended behavior.

```latex
def wall = 
\{
  \text{message} : m, \quad \text{incLikes} : like m.id
\} \mid m \leftarrow \text{messages}
```

This definition produces a collection that contains a message and an `action` value, for each row. Each action is a closure that contains the identifier for the message in the same row. By clicking on the “thumbs-up” icon, one triggers the execution (do) of the action, and thus increments the corresponding counter. So, the list of messages shown in the user interface in Figure 3 corresponds to accessing name `wall`.

Whenever the state of the application is changed, the collection defined by name `wall` is updated, and the new values can be pushed to the user interface. In summary, by adding a new message or clicking a “thumbs-up” icon of a message, the state variable `messages` is modified, causing the propagation of changes to the names that depend on it (i.e. `size` and `wall`), thus refreshing them. We think that this illustrates the semantic separation between the reactive and imperative parts of our language.

### 4. Programming Language

Our programming language, presented in Figure 4, consists in top-level declaration and interaction operations (O), and a \( \lambda \)-calculus as its functional core (e). A program \( \mathcal{P} \) is a sequence of top-level operations \( \mathcal{O} \) that is evaluated with relation to a running instance, enclosing state and code. We assume that \( a, b, c, \ldots \) range over a set of names \( \mathcal{N} \), and that \( x, y, z, \ldots \) range over a set of variables \( \mathcal{V} \). Names are globally visible in the application context providing a common namespace for all programming elements. Operations include the declaration/redefinition of state variables \( \text{var} a = e \) in the application name space, associating to name \( a \), the value denoted by expression \( e \). Another kind of operation is the declaration/redefinition of pure data transformation elements \( \text{def} a = e \) which associates name \( a \) to the value denoted by expression \( e \). Expressions defining names may use previously declared names. Finally, operations of the form \( \text{do} e \) represent the explicit execution of actions denoted by the expressions \( e \).

The functional core (e) includes abstraction \( \lambda x : \tau. e \) and application \( e' \) following call-by-value evaluation. For the sake of simplicity we assume the usual sets of base values \( b \) (i.e. strings, integers, booleans, etc.) and corresponding binary operators (op), and the ternary conditional operator (?). Names (a) in expressions are implicitly coerced to their denotations in the application state. We extend the base \( \lambda \)-calculus with collections, with constructor \( [\ldots] \), and corresponding concatenation (\( :: \)) and append (\( :: \)) operations. The iterator \( \text{iter}(e, e', x.y.e'') \) denotes a fold-left operation on the collection denoted by expression \( e \), and the iterated expression \( e'' \). Variable \( x \) denotes the current element in the collection, and variable \( y \) denotes either the value of expression \( e'' \) in the previous iteration, or the initial value given by expression \( e' \) in the first iteration. The destructor expression for collections \( \text{match} e \) with \( \text{xs} \rightarrow e' \) \( \{ \ldots \} \rightarrow e'' \) denotes one of two cases, depending on the value of expression \( e \). In the case of a non-empty collection, the evaluation proceeds with expression \( e' \), where variable \( x \) denotes the head of the collection, and variable \( \text{xs} \) denotes its tail. In the case of an empty collection (\( [] \)), the result is denoted by expression \( e'' \). An expression `action` \( \{ a := e \} \) value contains a finite sequence of delayed assignment statements to state variables. Note that a sequence of delayed statements may also be empty, in that case we write, `action` \( \{ \ldots \} \).

In the example presented in Section 3 we have extended this base language with extra orthogonal constructs, such as records and let declarations. We also used a list comprehension query notation, similar to the one introduced by Wadler [21]. List comprehension expressions can directly encoded in our core language in the usual way:

\[
\begin{align*}
& [e ~|~ x \leftarrow e'] \\
& [e ~|~ x \leftarrow e', f \text{ x}] \\
\end{align*}
\]

#### 4.1 Operational Semantics

We define a reactive operational semantics by means of two layers of small-step reduction relations, one for program operations, and another for \( \lambda \)-calculus expressions. The first layer is defined on program configurations \((\mathcal{S}, \mathcal{P}; \mathcal{L})\), where \( \mathcal{S} \) is a state mapping names (a) to triples with the form \((e, o, s)\), with \( e \) an expression, whose free names range over the domain of \( \mathcal{S} \), and \( o \) the current denotation of the name \( a \), that can be either a computed value (v), or undefined (\( \psi \)). The set of names \( s \) stores the names whose definition depend on \( a \), and thus need to be updated when the denotation of \( a \) changes. We call it the set of subscribers of \( a \). We write \( \mathcal{S}[a \mapsto (e, o, s)] \) to denote a state based on \( \mathcal{S} \) where \( a \) is associated with triple \((e, o, s)\).

The runtime values are defined by the fragment

\[
v ::= b ~|~ x ~|~ \lambda x : \tau. e ~|~ [v_1, \ldots, v_n] ~|~ \text{action} \{ a := e \}
\]

that includes a set of base values (including true and false), variables, \( \lambda \)-abstractions, and collections of values. Expressions of the form `action` are also values, containing a sequence of delayed imperative statements. The evaluation of the enclosing expressions, and the execution of the assignment statements is only triggered at the program level by a do operation. A program `P` in configuration \((\mathcal{S}, \mathcal{P}; \mathcal{L})\), represents the sequence of top-level operations yet to be applied to the running system, and queue \( \mathcal{L} \) denotes the queue of names that are scheduled to be updated, i.e. whose value needs to be propagated through the data-flow graph. We write \( [\ldots] \) to denote an empty queue, and use the notation \( \mathcal{L} + \mathcal{L}' \) for concatenation, and constructor \( \psi : \mathcal{L} \) to denote a queue with head \( a \) and a tail \( \mathcal{L} \).

The reduction relation on program configurations, written

\[
(\mathcal{S}; \mathcal{P}; \mathcal{L}) \rightarrow (\mathcal{S}'; \mathcal{P}'; \mathcal{L}')
\]

is defined by the rules in Figure 5. It is based on a reduction relation on expression configurations \((\mathcal{S}; e)\), written

\[
\mathcal{S}; e \rightarrow e'
\]

that specifies that expression \( e \) reduces to \( e' \) with relation to a state \( \mathcal{S} \). This reduction relation is defined by the rules shown in Figure 6.

Consider the following auxiliary abbreviations and definitions used in the program reduction relation, and essential to understand the rules in Figure 5.

**Definition 1 (Subscribers of a Name).** The set of subscribers of name \( a \), with relation to state \( \mathcal{S} \), written \( \psi(\mathcal{S}, a) \), is defined by

\[
\psi(\mathcal{S}, a) \triangleq \begin{cases} 
  s & \text{if } \mathcal{S}(a) = (e, o, s) \\
  \emptyset & \text{otherwise}
\end{cases}
\]
s = ψ(S, a) (VAR)

(S; var a = e, P; []) → (S[a → (e, s, s)]; P; []) (RDO)

S′ = subscribe(S, a, e)

(S; def a = e, P; []) → (S′[a → (e, s, s)]; P; []) (RDEF)

s = ψ(S, a)

(S[a → (e′′, o, s)]; do action { a := e; a′ := e′; P; [] }) → (S[a → (e, s, s)]; do action { a′ := e′; P; [a] }) (RASSIGN)

Figure 5. Program Operational Semantics.

Figure 6. Expression Operational Semantics.

Definition 2 (Free Names in Expressions). The set of free names of expression e, written σ(e), is defined by

σ(a) \triangleq \{ a \}

σ(b) \triangleq \emptyset

σ(x) \triangleq \emptyset

σ(λx.e) \triangleq σ(e)

σ(e e′) \triangleq σ(e) ∪ σ(e′)

σ(e op e′) \triangleq σ(e) ∪ σ(e′)

σ(e? e′ : e″) \triangleq σ(e) ∪ σ(e′) ∪ σ(e″)

σ(\langle e_{1}, \ldots, e_{n} \rangle ) \triangleq σ(e_{1}) ∪ \ldots ∪ σ(e_{n})

σ(\text{action } \{ \text{σ} \}) \\triangleq \emptyset

σ(\text{iter}(e,e′,x,y,e′′)) \\triangleq σ(e) ∪ σ(e′) ∪ σ(e′′)

σ(\text{match } e \text{ with } x:xs \rightarrow e | [] \rightarrow e′) \\triangleq σ(e) ∪ σ(e′) ∪ σ(e′′)

Notice that the set of free names of imperative statements enclosed in action values is always empty. Since the set of free names is the basic information used to build the data-flow graph, this expresses the fact that changes are not propagated into action values, and that their effects are not re-evaluated.

Definition 3 (Subscribe Function). The state, obtained from a state S, and updating the dependencies of a given name a, written subscribe(S, a, e), is defined by

subscribe(S, a, e) \triangleq

\{ b \rightarrow (e′, v, s \cup \{ a \}) | b \in \sigma(e) \land S(b) = (e′, v, s) \}

\cup \{ b \rightarrow (e′, v, s \setminus \{ a \}) | b \notin \sigma(e) \land S(b) = (e′, v, s) \}

The reduction relation on program configurations (Figure 5) is designed to sequentially evaluate the operations in program P, thus modifying state S and queue L accordingly. The general idea behind a program configuration is to use the queue L to express the reactive semantics, and signal the situations where a system reconfiguration can happen without disruption, i.e. where new operations can be evaluated. Rules VAR, RDO, and RASSIGN deal with the three top-level operations, and their basic effect is to change the definition of a given name in the state, and signal that it must be re-evaluated by placing it in the (empty) queue. Rule RQSTEP starts the process of updating the value of name in the front of the queue. This is accomplished by placing the associated expression at the queue’s head (a(e):L). Rule RQSTEP iteratively reduces the expression at the head of the queue, by using the reduction relation for expressions from Figure 6. Rule RQVAL updates the name at head of the queue with a computed value, and signals for the re-evaluation of all names whose definition depends on name a (i.e. its subscribers) by appending s to queue L. Notice that var, def and do operations only reduce in an empty queue, rules VAR, RDO, and RASSIGN.

Notice that in the case of a def operation, rule RDEF updates the initial state S so that all dependencies of a are refreshed (see above for the definition of subscribe(S, a, e)). In the case of a var operation there is no need to update the state with new subscribers, since changes are not propagated into state variable definitions.

Rule RDO reduces the subexpression e, in an operation do e, to an action value. Rule RASSIGN processes one assignment at a time. The effect of an assignment is achieved by overwriting the expression associated to the assigned name in the state, and by queuing the name for evaluation. Finally, rule RSKIP ignores the empty action value, and continues the execution of the remaining operations in P.

Our program operational semantics defines a data-driven (push) approach, in opposition to a pull approach of other reactive languages [12]. This results in the immediate propagation of changes through the dependency graph, until reaching an empty queue. In
Section 4.2 we present a type system that establishes enough conditions so that we can prove convergence of the propagation process (see Section 6). A halting configuration is reached when no operation is available and the queue is empty, i.e. it is of the form (S::[]):

We next present the reduction relation on expressions, defined by the rules in Figure 6, based on a standard notion of evaluation contexts (E) with rule RCONTEXT specifying the expression reduction in a context.

Definition 4 (Expressions Contexts). Evaluation contexts for expressions, E, are defined by

\[ E ::= \epsilon \mid v \cdot E \mid e \cdot E \mid e \cdot e' \mid [v_1, \ldots, v_n, E_1, E_2, \ldots, E_m] \mid \text{iter}(E, e, x, y, e') \mid \text{match} E \text{ with } x : x \cdot E \to e \mid [\ ] \to e' \]

Rule RNAME specifies the implicit dereference of a name, reducing it to the corresponding value in state S. Reduction of binary operations (ROP) is abstracted from the system, and rule RAPP implements the usual reduction of a call-by-value application. The ternary conditional operator (RTRUE and RFALSE) also follows the usual semantics. The semantics of collection iteration (RNIL and RITERATE), and collection destruction (REMPLOY and RMATCH) also follow straightforward semantics.

4.2 Type System

Our type language comprises a representative for basic types \( \beta \) (e.g. Bool, Int), function types \( \tau \to \tau' \), types for homogeneous collections (\( \tau^* \)), and type Action for action values.

\[ \tau ::= \beta \mid \tau \to \tau' \mid \tau^* \mid \text{Action} \]

The typing relation on programs is defined by a set of rules, shown in Figure 7, which directly depends on the typing relation on expressions and statements, in Figure 8. Both typing judgments are defined with relation to typing environments \( \Gamma \), that are mappings from variables to types, and names to special type annotations. Type annotations for names are designed to carry extra information about names, and are either of the form def(\( \tau \)) or var(\( \tau \)). In both cases, \( \delta \) is a set of names containing all name dependencies of the expression associated to the annotated name. This set becomes essential to avoid cyclic definitions, which would cause infinite loops during propagation of changes, and also allows to statically discipline the redefinition of names.

The typing relation of programs is defined by a judgment of the form \( \Gamma ; \delta \vdash P \), and is inductively defined by the rules in Figure 7. The typing of expressions has the judgment of the form \( \Gamma ; \delta \vdash e : \tau \), which asserts that expression \( e \) has type \( \tau \) with relation to the typing environment \( \Gamma \). An invariant of this judgment is that expression \( e \) uses, at most, the set of names \( \delta \), with \( \delta \subseteq \text{dom}(\Gamma) \). Finally, the typing of assignments is kept separate, with the judgment \( \Gamma ; \delta \vdash a := e \), that asserts that the assignment \( a := e \) is well-typed with relation to the typing environment \( \Gamma \), and uses, at most, the set of names \( \delta \). 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 statements are presented in Figure 8.

The typing rules for programs, in Figure 7, follow the general structure of sequentially examining the top-level operations in a program. The conditions to define new names are the same for state variables (VAR) and for data transformation expressions (DEF). At this stage we require that the declared name (a) is fresh in the typing environment (a \( \not\in \text{dom}(\Gamma) \)). This statically disallows the redefinition of names, which are covered in Section 5.1 by an extended version of the type system. The condition a \( \not\in \Gamma(\delta) \) considers all the names used in expression e, and its expansion with relation to the typing environment as defined below. This statically ensures that the declared name is not transitively used in the expression e.

Definition 5 (Extended Dependencies). The extended dependencies of a set of names \( \delta \), with relation to a typing environment, written \( \Gamma(\delta) \), is defined by:

\[ \Gamma(\delta) \triangleq \delta \cup \bigcup_{a \in \delta} \{ \Gamma(a) = \text{def}(\tau) \} \]

Computing the extended set of name dependencies is essential to statically avoid circular dependencies, and ensuring the convergence of the propagation of changes. Notice that the dependencies of the state variables in \( \delta \) are not expanded. Thus, they are not accounted for in the cycle detection condition. This is sound because, at runtime, changes are not propagated into the definitions of state variables. Recall that the initializer expressions of state variables are only evaluated once. Our formal results include a relation between a well-typed state and well-formed extended dependency set (see Appendix B). The typing of top-level operations of the form do e, covered by rule DO, just requires that expression e is typed with type Action. Notice that the dependencies of e (\( \delta \)) are ignored in the bookkeeping process. This is sound given that the effect of an action is never re-executed by the propagation of changes.

The typing relation on expressions, in Figure 8, follows standard lines and is extended so that name dependencies (\( \delta \)) are conservatively computed. The most interesting cases are the axioms for names (DNAME and VNAME), where resulting dependencies must include the name itself, and rule for assignment statements (ASSIGN) that requires the assigned name to be included in the dependencies. In all other cases, dependencies are simply propagated through the derivation.

5. Dynamic Reconfiguration

In this section we extend our language to support the dynamic reconfiguration. We propose a uniform mechanism that allows for the existing operations to redefine names, and extend our type system to ensure that all reconfigurations are sound. Consider the example from Section 3, which started with a simple data model and functionality, and we now want to evolve to meet a new requirement, that is to store which users have "liked" each message, instead of simply counting occurrences. In our dynamic reconfiguration model, it is possible to incrementally reconfigure the existing system. Figure 9 depicts the sequence of operations that are needed to meet this new requirement.

We start by creating a new state variable (whoLikes) to store the relation between messages and users. Note that, in this case, there was no previous information of which user liked each message, and for the sake of simplicity, we instantiate the state variable with only one record stating that user "Henry" "liked" the message with id 0.
the newly defined state variables messages message identifier. Finally, name number of “likes” for that particular message, by iterating over the new collection, instead of

Given the reconfigured persistent state, we now redefine the necessary variables and counting the occurrences with an equal size, and should be questioned in intermediate states of the reconfiguration. That problem is out of the scope of this paper, and should be addressed in future work, by means of orthogonal control mechanisms.

5.1 Typing Name Redefinition
We now describe the extension of our programming language to support the incremental reconfiguration of systems. Notice that the operational semantics already defined, in Section 4.1, already supports the redefinition of state variables and data transformation expressions. So, the proposed extension is defined at the type level, in order to disallow sequences of redefinitions that are not sound.

The typing rules introduced, in Section 4.2, require that type name definitions do not support any form of redefinition. Therefore, we present in Figure 10, additional rules that allow and discipline it. We divide the redefinition of programming elements (var and def) into three different cases.

In the first case, rules UVAR and UDEF, type the redefinition of a name, by allowing a different expression while maintaining the same type. These rules extend the ones presented in Section 4.2 by checking that the type is maintained, and that the old dependencies (δ) registered in Γ are replaced by the new dependencies (δ′), given by the typing of expression e. We prove that this is enough information to continue avoiding circular name dependencies. We
also show that, since the type is not modified, all expressions in the state are still sound.

The second case is covered by rules **UVART** and **UDEFT**, that type the reconfiguration of a name using an expression of a different type. Both rules ensure that a name can change type only if no other name is using it. This is ensured by the condition \( a \not\in \rho(\Gamma) \), where \( \rho(\Gamma) \) stands for the union of all dependencies sets in \( \Gamma \).

**Definition 6** (Union of Name Dependencies). The union of the name dependencies sets, in a given a typing environment \( \Gamma \), written \( \rho(\Gamma) \), is defined by

\[
\rho(\Gamma) \triangleq \bigcup_{a \in \text{dom}(\Gamma)} \{ \delta \mid \Gamma(a) = \text{var}_\delta(\tau) \lor \Gamma(a) = \text{def}_\delta(\tau) \}
\]

With this extra condition, we disallow the redefinition of a name that is being used elsewhere in the system. Reconfigurations, that change type specification, are possible through sequences of reconfiguration operations that are performed in such an order that no broken dependencies ever exist in the application.

Finally, the third allowed case, is expressed by rule **VARDEF**, where there is a modification from a state variable to a pure data transformation element. For the sake of simplicity we disallow the dual case, which would require recomputing of all name dependencies in \( \Gamma \).

In summary, reconfiguration is supported in a disciplined and ordered way. If maintaining the type, cycles are still detected. And, if the type is modified then a specific order must be followed to ensure that no definition is broken.

### 6. Type Safety

The soundness results for our language are based on proving type safety and change propagation convergence, which we derive from standard subject reduction theorems. Beside the usual type errors, our type system also ensures that the intended (reactive) behavior is kept sound even when the system is reconfigured. We have limited the expressiveness of the functional core to be able to formally establish the termination of the propagation process. Nevertheless, the language can be orthogonally extended to include general recursion. The results presented here follow a syntactic approach [24], and are proved by induction on the length of the typing derivations. We provide proof sketches for most results, whereas the full proofs can be found in Appendix B. All results are based on the typing of runtime program configurations, which in turn relies on a well-formed relation on queues, and well-typed relations on states and programs.

The notion of well-formed queue (Definition 7) with relation to a state and typing environment, written \( S; \Gamma \vdash L \), asserts that the partial order imposed by the subscribers relation in \( S \) is well-founded. We establish that the empty queue is always reachable from queue \( L \), through the subscriber relation. Also, when a queue is of the form \( a(e):L \), we establish that expression \( e \) has the same type according to the type annotation of \( a \) in the typing environment \( \Gamma \). We prove that \( S; \Gamma \vdash \mathcal{L} \) is an invariant of the reduction relation, and conclude that the update process converges (see Appendix B).

**Definition 7** (Well-Formed Queue). A queue \( L \) is well-formed with relation to a state \( S \) and a typing environment, written \( S; \Gamma \vdash L \), if it can be inductively defined by the rules

\[
\frac{\Gamma; \delta \vdash e : \tau \quad a \not\in \bigcap_\delta(\delta) \quad \tau \neq \tau'}{\Gamma; a : \text{var}_\delta(\tau) \vdash e \sigma e, P} \quad (\text{UVART})
\]

\[
\frac{\Gamma; \delta \vdash e : \tau \quad a \not\in \bigcap_\delta(\delta) \quad \tau \neq \tau'}{\Gamma; a : \text{def}_\delta(\tau) \vdash e \sigma e, P} \quad (\text{UDEFT})
\]

\[
\frac{\Gamma; \delta \vdash e : \tau \quad a \not\in \bigcap_\delta(\delta) \quad \tau \neq \tau'}{\Gamma; a : \text{def}_\delta(\tau') \vdash e \sigma e, P} \quad (\text{VAREDF})
\]

The typing judgment for states (Definition 9), written \( \Gamma \vdash \mathcal{S} \), asserts that the names in state \( S \) and its subscribers form a well-founded partial order in \( \Gamma' \), and all expressions contained in \( S \) are well-typed with relation to the typing environment \( \Gamma' \), \( S'. \) The typing relation imposes the disjointness of \( \text{dom}(S) \) and \( \text{dom}(\Gamma') \), and also a well-founded order on names, through name dependency relation contained in typing environment \( \Gamma' \), in a structure resembling the ordered logic approach [16]. This separation of the typing environment (\( \Gamma \) and \( \Gamma' \)) establishes the absence of circular dependencies, since subscribers (\( s \)) are on one side, and name dependencies (\( \delta \)) on the other side. On the one hand, the expression associated to a name in state \( S \) is only dependent on names from the typing environment \( \Gamma \) i.e. \( \bigcap_\delta(\delta) \subseteq \text{dom}(\Gamma) \). On the other hand, a name is only used by expressions associated to names on \( \Gamma' \), \( \bigcup_{a \in \mathcal{S}} \subseteq \text{dom}(\Gamma') \), where \( \mathcal{S} = \bigcup_{a \in \mathcal{S}} \) is the expansion of the subscribers relation in \( S \) (Definition 8). From these invariant conditions, we conclude that for any name \( a \) in a well-typed state \( S \), the singleton queue \( [a] \) is always well-formed, \( S, a \rightarrow (e, a, s); \Gamma' \vdash [a] \). Note that the names remaining in \( S \) are the same names remaining in \( \Gamma' \) (both domains are equal).

**Definition 8** (Expansion of Subscribers). The expansion of subscribers relation, written \( \bigcup_{a \in \mathcal{S}} \), for a set of names \( s \), with relation to a typing environment \( S \), is defined by

\[
\bigcup_{a \in \mathcal{S}}(s) \triangleq s \cup \bigcup_{a \in \mathcal{S}} \{ \bigcup_{a \in \mathcal{S}}(s') \mid s' = \psi(S, a) \}
\]

**Definition 9** (Well-Typed State). A state \( S \) is well-typed with relation to two typing environments \( \Gamma' \) and \( \Gamma'' \) with \( \text{dom}(\Gamma') \cap \text{dom}(\Gamma'') = \emptyset \), if \( \Gamma' \vdash \mathcal{S} \) can be inductively defined by the rules in Figure 11.

We next introduce the notion of well-typed program configuration (Definition 10), which relies on the previous definitions of well-typed state (Definition 9), well-typed program (presented in Section 4.2), and well-formed queue (Definition 7).

**Definition 10** (Well-Typed Program Configuration). A program configuration, written \( (S; P; \mathcal{L}) \), is well-typed with relation to a
typing environment $\Gamma$, if $\Gamma \vdash (S; P; L)$ is derivable by the following rule

\[
\Gamma \vdash S \quad \Gamma \vdash P \quad \Gamma \vdash L \\
\Gamma \vdash (S; P; L)
\]

We also define a well-typed expression configuration (Definition 11), relying on the previous definitions of well-typed state (Definition 9), and well-typed expression (presented in Section 4.2).

Definition 11 (Well-Typed Expression Configuration). An expression configuration, written $(S; e)$, is well-typed with relation to a typing environment $\Gamma$, if $\Gamma \vdash (S; e)$ is derivable by the following rule

\[
\Gamma \vdash S \quad \Gamma \vdash e : \tau \\
\Gamma \vdash (S; e)
\]

Based on these auxiliary definitions, we establish type safety and present here the main results for programs: progress (Theorem 1), type preservation (Theorem 2) and queue convergence (Theorem 3). These results are based on intermediate results for expressions, states, and queues. For full details and proofs, please refer to Appendix B.

To establish progress of programs (Theorem 1), we need to ensure a stronger invariant of the reduction relation, that says that no names can be undefined (□), except perhaps for the name currently being evaluated, i.e. if there is an undefined value in the state, it corresponds to the name which is the only element in the evaluation queue.

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

Type preservation for programs (Theorem 2) follows standard lines, and is proved by induction on the derivation of $(S; P; L) \rightarrow (S'; P'; L')$. 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 RDEF, RVAR, which involve an inversion property in the typing of states, and RQVAR, where we need to consider the cases of top-level definitions and variables, and carefully reconstruct well-typed states and well-formed queues.

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

In order to prove the convergence of the evaluation queue, we introduce a measure on names (Definition 12) 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 program configurations always reach an empty queue.

Definition 12 (Name Length). We define the length of a name $a$ with relation to a state $S$, written $m_a(S)$, as the sum of the lengths of all its names (Definition 13).

Definition 13 (Queue Length). We define the length of a queue $L$ with relation to a state $S$, written $m_L(S)$, as the sum of the lengths of all its names.

Intuitively, the length of a name represents the number of computational steps needed to propagate an update through the name dependencies 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 length function for a particular queue. This results in a lemma (Lemma 1) establishing that a non-empty queue decreases in length after a finite number of reduction steps.

Lemma 1 (Queue Convergence). For all configurations $(S; P; L)$ and $(S'; P'; L')$, and typing environments $\Gamma$, if $\Gamma \vdash (S; P; L)$ and $L \neq []$ then $(S; P; L) \rightarrow^* (S'; P'; L')$ and $m_L(S) < m_L(L')$.

Based on this previous result, we can now establish a stronger result (Theorem 3) where a well-typed program configuration with a non-empty queue, reaches a program configuration with an empty queue after a finite number of steps.

Theorem 3 (Convergence). For all configurations $(S; P; L)$ and $(S'; P'; L')$, and typing environments $\Gamma$, if $\Gamma \vdash (S; P; L)$ then $(S; P; L) \rightarrow^* (S'; P'; [\cdot])$.

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 change propagation (Theorem 3). These results allow us to ensure that the automatic propagation of changes caused by the execution of imperative assignments.
7. Extending Our Example

In this section we extend our bulletin board application previously introduced in Section 3. This extension adds a filtering mechanism that allows to show only the messages of a given user, e.g. by clicking on the user’s name in the UI (Figure 13(a)) would display only the messages from that user (Figure 13(b)). Notice that this extension may be applied after or before the reconfiguration introduced in Section 5 (Figure 9), obtaining the same final application.

Consider the code depicted in Figure 12. We start by creating the noFilter name, that is one of our filters. In this case, it acts like “show all” filter. We use the state variable currentFilter to hold the filter currently applied to the list. This allows us to modify the active filter with the execution of an action. Name filteredMessages iterates over the messages collection, filtering the messages according to the active filter (currentFilter). Notice that this new name, has the same behavior as name messages defined in Figure 2 and then redefined in Figure 9. At this point, none of the messages is filtered, since the active filter corresponds to the noFilter, which returns true for any given message m. This results in a list of filtered messages containing all the messages. To achieve our goal of filtering messages by author we define name authorFilter. We also reconfigure our application by redefining name wall to use the filteredMessages list, instead of messages. Hence, this reconfiguration displays messages according to the active filter (currentFilter).

In Figure 13 we show the possible states of the active filter. By clicking on the author’s name associated to each message, the active filter changes, and so does the list of messages shown. For instance, clicking on “Paul” corresponds to the execution of

\[
\text{def noFilter} = \lambda m. \text{true}
\]

\[
\text{var currentFilter} = \text{noFilter}
\]

\[
\text{def filteredMessages} = \{ m \mid m \leftarrow \text{messages}, \text{currentFilter} m \}
\]

\[
\text{def authorFilter} = \lambda a. \lambda m. (m.\text{author} = a)
\]

\[
\text{def wall} = \{ \{ \text{message} : m, \text{incLikes} : \text{likes} m.\text{id} \} \mid m \leftarrow \text{filteredMessages} \}
\]

Figure 12. Message Filters Code.

happens without causing infinite propagation loops, and dynamic reconfiguration of both code and data does not break type safety.

8. Related Work

Data-flow programming languages have been extensively studied in the past and summarized in surveys [12]. The pure data-flow model considers that every node is pure and without side effects. Likewise, our definition of a data-flow graph, obtained from all top-level operations, def or var, is built out of effect-free expressions. Imperative features are kept apart from the data-flow propagation mechanism. Many data-flow programming languages also explore parallel computation by computing several non-interfering nodes simultaneously. We focus on the core reconfiguration model, and we believe that such concerns, although interesting, are orthogonal. The support to parallel propagation can be achieved with an execution queue, sensitive to data dependencies.

Adaptive Functional Programming [1] uses an underlying dependency graph and a change propagation algorithm to adapt to input changes. The change propagation algorithm is encoded into traces in the operational semantics. We encode a similar propagation mechanism into our operational semantics, by means of the evaluation queue. Our approach, although inspired in some aspects on AFP, focuses on providing a terminating reactive framework that can be dynamically changed.

FrTime [7] is another approach for writing functional reactive applications. Reactivity is embedded into a call-by-value dynamic programming language. In principle, our language can be encoded into the FrTime’s reactive framework, where cells play the role of state variables (var), pure data transformation elements (def) can be mapped to computations over signals, and execution (do) of actions can be encoded to the set-cell! operator in FrTime. However, our approach goes beyond that point, by statically verifying programs, to ensure convergence of change propagation, and also the safety of reconfigurations. Unlike our language, FrTime allows that a cyclic, unguarded, definition is added to the system, causing it to diverge.

Reactive systems [9, 14, 15, 17] maintain an ongoing interaction with their environment, and 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 actions, causing the system to react. Our incremental style of programming also maintains an ongoing interaction with the environment, where reconfigurations can be issued, or actions can be executed to change the application persistent state. Imperative reactive languages [3, 4, 10] usually 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 [14, 17, 22, 23]. However, neither of these approaches combines reactivity with a statically verified dynamic reconfiguration mechanism as the one presented in this paper.

TouchDevelop [5] provides a programming environment with immediate feedback in the development of user interfaces. It provides a statically verified approach by means of a type and effect system, that separates presentation and application code. The edit-compile-run development cycle is tightened by allowing the display code to be refreshed without restarting programs. Our approach provides a more uniform programming mechanism that also allows the evolution of complete programs without the need to recompile or restart it.
Several works on dynamic reconfiguration and incremental computation have already been proposed [8, 11, 20], and are mostly designed for imperative programming languages. For instance [11] provides dynamic updates in programs by explicitly defining logical regions where updates may occur. When the execution of a program reaches an update region, if a dynamic update is available at that point in the execution, it is applied. Up to some point, our approach is similar. We do not explicitly define update regions or execution points, but we interpolate reconfiguration actions with each update cycle of the system. By combining reactivity with dynamic reconfiguration we are able to maintain a name dependency graph that changes dynamically, while maintaining the typing invariants. To the best of our knowledge, this paper presents the first work that supports dynamic reconfiguration in reactive and imperative programming languages.

9. 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. We have also stated provable type preservation and progress theorems for programs, including a queue convergence result. Besides supporting dynamic reconfiguration, our operational semantics also supports reactivity by propagating changes through a name dependencies graph. This means that visible names, are always up-to-date with respect to the persistent state of the application. Our type system statically ensures that both data propagation and reconfiguration happens without breaking type safety.

We believe that this is a suitable core model, at the right abstraction level, for an agile development tool that assists on the development of data-centric reactive applications requiring constant evolution. We can expect high productivity gains provided by such a programming environment that, at each construction step and reconfiguration action, ensures the consistency of code and underlying data, and provides constant feedback about the new added data and behavior. Our future roadmap includes extensions related to control sharing to support collaborative developments. Other interesting features involve parallelization of the queue based on separation properties, and the obvious optimizations of avoiding unnecessary computations.

References
A. Extra Definitions

Definition 14 (Substitution). Given an expression \( e \), a value \( v \) and a variable \( x \), the substitution relation for expression, written \( e \{ v/x \} \), replaces all occurrences of \( x \) by the value \( v \) in \( e \). We define the substitution relation for expressions inductively, as follows

\[
\begin{align*}
x \{ v/x \} & \triangleq v & (x \neq y) \\
y \{ v/x \} & \triangleq y \\
a \{ v/x \} & \triangleq a \\
\text{(action \{ } a := e \text{ }) \{ v/x \} & \triangleq \text{(action \{ } a := e \{ v/x \} \text{ }) \\
\lambda x. e \{ v/y \} & \triangleq \lambda x. (e \{ v/y \}) & (x \neq y) \\
v \{ x/v \} & \triangleq v & (v \neq \lambda x.e \land v \neq \text{action \{ } a := e \text{ }) \\
(e f) \{ v/x \} & \triangleq (e \{ v/x \}) (f \{ v/x \}) \\
(e \op f) \{ v/x \} & \triangleq (e \{ v/x \}) \op (f \{ v/x \}) \\
(e \op f) \{ v/x \} & \triangleq (e \{ v/x \}) ? (e' \{ v/x \}) : (e'' \{ v/x \}) \\
[e_1, \ldots, e_n] \{ v/x \} & \triangleq [e_1 \{ v/x \}, \ldots, e_n \{ v/x \}] \\
\text{iter}(e, f, x, y, g) \{ v/x \} & \triangleq \text{iter}(e \{ v/z \}, f \{ v/z \}, x, y, g \{ v/z \}) & (z \neq x \land z \neq y) \\
\text{match } e \text{ with } y::y \rightarrow f : g \{ v/x \} & \triangleq \left( \text{match } e \{ v/x \} \text{ with } y::y \rightarrow f \{ v/x \} : (g \{ v/x \}) \right) & (x \neq y)
\end{align*}
\]
B. Proofs

Lemma 2 (Substitution for Expressions). For all expression configurations $e$ and typing environments $\Gamma$, if $\Gamma, x \vdash \tau'; \delta \vdash e : \tau$ and a value $v$ such that $\Gamma; \delta \vdash v : \tau'$, then $\Gamma; \delta \vdash e\{v/x\} : \tau$.

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

1. Base Value
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash b' : \tau$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $b'\{b/x\} = b'$
   * $\Gamma; \delta \vdash v' : \tau$
      by Definition 14
      from (H1, 3)

2. Variable Substitution
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash x : \tau$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $\tau = \tau'$
   (4)  $x\{v/x\} = v$
   (5)  $\Gamma; \delta \vdash v : \tau$
      by Definition 14
      from (H2, 3, 4)

3. Variable $x \neq y$
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash a : \tau$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $a\{v/x\} = a$
   (4)  $\Gamma; \delta \vdash y : \tau$
      by Definition 14
      from (H1, 3)

4. Name
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash a : \tau$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $\tau = \text{var}(\tau'') \lor \tau = \text{def}(\tau'')$
   (4)  $\Gamma; \delta \vdash a : \tau$
      with $\tau = \text{var}(\tau'') \lor \tau = \text{def}(\tau'')$
      by inv. of ACTION in (H1)

5. Action
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash \text{action} \{ a := \tau \} : \text{Action}$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $\text{action} \{ a := \tau \}\{v/x\} = \text{action} \{ a := e\{v/x\} \}$
   (4)  $\delta = \delta' \cup \{ a \}$
   (5)  $\Gamma; \delta' \vdash e_i\{v/x\} : \tau_i$
   (6)  $\Gamma; \delta \cup \{ a \} \vdash a_i := e_i\{v/x\}$
   (7)  $\Gamma; \delta \vdash \text{action} \{ a := e\{v/x\} \} : \text{Action}$
      by inv. of ACTION in (H1)
      by L.H. with (5, H2)

6. Abstraction
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash \lambda y.e : \tau_y \rightarrow \tau_e$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $x \neq y$
   (4)  $\Gamma; y : \tau_y; \delta \vdash e\{v/x\} : \tau_e$
   (5)  $\Gamma; \delta \vdash \lambda y.(e\{v/x\}) : \tau_e$
      by Definition 14
      by inv. of ABS in (H1)
      by L.H. with (3, H2)

7. Application
   (H1)  $\Gamma, x \vdash \tau'; \delta \vdash e e' : \tau$
   (H2)  $\Gamma; \delta \vdash v : \tau'$
   (3)  $(e e')\{v/x\} = (e\{v/x\}) (e'\{v/x\})$
   (4)  $\Gamma; x : \tau'; \delta \vdash e : \tau_a \rightarrow \tau$
   (5)  $\Gamma; \delta \vdash e\{v/x\} : \tau_a \rightarrow \tau$
   (6)  $\Gamma; \delta \vdash e'\{v/x\} : \tau_a$
   (7)  $\Gamma; \delta \vdash (e\{v/x\}) (e'\{v/x\}) : \tau$
      by Definition 14
      by inv. of APP in (H1)
      by in. of APP in (H1)
      by L.H. with (H2, 3)
      by L.H. with (H2, 4)
      by APP with (5, 6)

8. Binary Operation
   (H1)  $\Gamma, x \vdash \tau''; \delta \vdash e \text{ op } e' : \tau''$
   (H2)  $\Gamma; \delta \vdash v : \tau''$
   (3)  $(e \text{ op } e')\{v/x\} = (e\{v/x\}) \text{ op } (e'\{v/x\})$
      by Definition 14
Proof. By induction on the length of the derivation \( \Gamma; \delta \vdash e : \tau \).

1. **DNAME**

   \( \Gamma \vdash (S; a) \)

2. \( \Gamma, a : \text{def}_{S}(\tau); \delta \cup \{a\} \vdash a : \tau \)

3. \( S(a) = (e, v, s) \)

   \( S; a \rightarrow v \)

   by **RNAME** with (H3)
2. VNAME
(H1) \( \Gamma, a : \text{var} \delta(\tau); \delta \cup \{a\} \vdash a : \tau \)
(H2) \( \Gamma \vdash (S; a) \)
(H3) \( \Delta(a) = (e, v, s) \)
* \( \Delta; a \rightarrow v \)

3. ID
(H1) \( \Gamma; \delta \vdash x : \tau \)
(H2) \( \Gamma \vdash (S; x) \)
(H3) \( \forall a \in \delta, \Delta(a) = (e, v, s) \)
* \( x \)

4. ABS
(H1) \( \Gamma; \delta \vdash \lambda x.e : \tau \rightarrow \tau' \)
(H2) \( \Gamma \vdash (S; \lambda x.e) \)
(H3) \( \forall a \in \delta, \Delta(a) = (e, v, s) \)
* \( \lambda x.e \)

5. APP
(H1) \( \Gamma; \delta \vdash e_1 e_2 : \tau' \)
(H2) \( \Gamma \vdash (S; e_1 e_2) \)
(H3) \( \forall a \in \delta, \Delta(a) = (e, v, s) \)
* \( \Delta; e_1 e_2 \rightarrow e_1 \)

6. APP
(H1) \( \Gamma; \delta \vdash \text{op} e_2 : \tau'' \)
(H2) \( \Gamma \vdash (S; e_1 \text{op} e_2) \)
(H3) \( \forall a \in \delta, \Delta(a) = (e, v, s) \)
* \( \Delta; e_1 \text{op} e_2 \rightarrow e_1 \text{op} e_2 \)

7. IF
(H1) \( \Gamma; \delta \vdash (e \ ? \ e' : e'') : \tau' \)
(H2) \( \Gamma \vdash (S; e \ ? \ e' : e'') \)

by rname with (H3)

is a value

by inv. of APP in (H1)

by inv. of APP in (H1)

by Definition 11 with (H2)

by Definition 11 with (6, 4)

by I.H. with (7, 4, H3)

by rcontext with (8)

by Definition 11 with (6, 5)

by I.H. with (11, 5, H3)

by rcontext with (10, 12)

from (4, 8)

by rapp with (8, 14)

by rcontext with (8)

by Definition 11 with (6, 5)

by I.H. with (11, 5, H3)

by rcontext with (10, 12)

by rop with (10, 14)
∀a ∈ δ. S(a) = (e, v, s) by Definition 11 with (H2)

Γ ; δ ⊢ e : Bool by inv. of it in (H1)

Γ ⊢ (S; e)

S ; e → e1 ∨ e is a value by I.H. with (6, 5, H3)

CASE: S ; e → e1
  ∗ S; (e ? e′ : e″) → (e1 ? e′ : e″) op e2 by RCONTEXT with (7)

CASE: e is a value
  e = true ∨ e = false from (5, 9)

SCASE: e = true
  ∗ S; (true ? e′ : e″) → e′ by RTRUE with (10)

SCASE: e is false
  ∗ S; (false ? e′ : e″) → e″ by RTRUE with (12)

8. ITER

[H1] Γ ; δ ⊢ ite(r1, x, y, e1) : τ′

[H2] Γ ⊢ (S; ite(r1, x, y, e1))

[H3] ∀a ∈ δ. S(a) = (e, v, s)

Γ ; δ ⊢ e : τ′ by inv. of ite in (H1)

Γ ; δ ⊢ e2 : τ′ by inv. of ite in (H1)

Γ ; x : τ ; y : τ′ ; δ ⊢ e3 : τ′ by inv. of ite in (H1)

S ; e1 → e′1 ∨ e1 is a value by I.H. with (4, 8, H3)

CASE: S ; e1 → e′1
  ∗ S; ite(r1, x, y, e1) → ite(r1, x, y, e1) by RCONTEXT with (9)

CASE: e1 is a value
  Γ ⊢ (S; e2)
  S; e2 → e′2 ∨ e2 is a value by I.H. with (12, 5, H3)

SCASE: S; e2 → e′2
  ∗ S; ite(r1, x, y, e1) → ite(r1, x, y, e1) by RCONTEXT with (11, 13)

SCASE: e2 is a value
  e1 = [] ∨ e1 = [x1, ..., xn] from (4, 11)

SSCASE: e1 = []
  ∗ S; ite(r1, x, y, e1) → e2 by RILE with (16, 15)

SSCASE: e1 = [x1, ..., xn]
  ∗ S; ite(r1, x, y, e1) → ite([x2, ..., xn], e3{x1/x}{e2/y}, x, y, e3) by RITERATE with (18, 15)

9. MATCH

H1) Γ ; δ ⊢ match e1 with x : xs → e2 | [] → e3 : τ′

H2) Γ ⊢ (S; match e1 with x : xs → e2 | [] → e3)

H3) ∀a ∈ δ. S(a) = (e, v, s)

Γ ; δ ⊢ e1 : τ′ by inv. of MATCH in (H1)

Γ ; x : τ ; xs : τ′ ; δ ⊢ e2 : τ′ by inv. of MATCH in (H1)

Γ ; δ ⊢ e3 : τ′ by inv. of MATCH in (H1)

S ; e1 → e1′ ∨ e1 is a value by I.H. with (8, 4, H3)

CASE: S ; e1 → e1′
  ∗ S; match e1 with x : xs → e2 | [] → e3 → match e1′ with x : xs → e2 | [] → e3 by RCONTEXT with (9)

CASE: e1 is a value
  e1 = [] ∨ e1 = [x1, ..., xn] from (11, 4)

SCASE: e1 = []
  ∗ S; match e1 with x : xs → e2 | [] → e3 → e3 by REMPTY with (12)

SCASE: e1 = [x1, ..., xn]
10. COLLECTION
(H1) \( \Gamma; \delta \vdash [e_1, \ldots, e_n] : \tau^* \)
(H2) \( \Gamma \vdash (S; [e_1, \ldots, e_n]) \)
(H3) \( \forall a \in \delta.S(a) = (e, v, s) \)
(4) \( \Gamma; \delta \vdash e_i : \tau \)
(5) \( \cdot \Gamma \vdash S \)
(6) \( \Gamma' \vdash (S; e_i) \)
\( S; e_i \rightarrow e_i' \vee e_i \) is a value
(7) CASE: \( S; e_i \rightarrow e_i' \)
\( e_0, \ldots, e_{i-1} \) are values
\( S;' [e_1, \ldots, e_2, \ldots, e_n] \rightarrow [e_1, \ldots, e_i', \ldots, e_n] \)
by \( \text{RCONTEXT} \) with (7)
(9) CASE: \( e_i \) is a value
\( e_0, \ldots, e_n \)
\( [e_1, \ldots, e_n] \)
are values
is a value
11. VALUE
(H1) \( \Gamma; \delta \vdash b : \beta \)
(H2) \( \Gamma \vdash (S; b) \)
(H3) \( \forall a \in \delta.S(a) = (e, v, s) \)
* \( b \)
is a value
12. ACTION
(H1) \( \Gamma; \delta \vdash \text{action} \{ \overline{a} := \tau \} : \text{Action} \)
(H2) \( \Gamma \vdash (S; \text{action} \{ \overline{a} := \tau \}) \)
(H3) \( \forall a \in \delta.S(a) = (e, v, s) \)
* \( \text{action} \{ \overline{a} := \tau \} \)
is a value

Lemma 4 (Expressions Context Type Preservation). For all expression configurations \( (S; \mathcal{E}[e]) \) and typing environments \( \Gamma \), if \( \Gamma; \delta \vdash e : \tau \) and \( \Gamma; \delta \vdash e' : \tau' \), then \( \Gamma; \delta \vdash \mathcal{E}[e'] : \tau \).

Proof. Straightforward by induction on the structure of \( \mathcal{E}[e] \).

Lemma 5 (Expressions Type Preservation). For all expression configurations \( (S; e) \) and typing environments \( \Gamma \), if \( \Gamma \vdash (S; e) \) and \( \Gamma; \delta \vdash e : \tau \) and an expression reduction \( S; e \rightarrow e' \), then \( \Gamma; \delta \vdash e' : \tau \).

Proof. By induction on the length of the expression reduction \( S; e \rightarrow e' \).
(20) \( \Gamma; \delta \vdash e : \tau' \rightarrow \tau \)
(21) \( \Gamma \vdash (S, e) \)
(22) \( \Gamma; \delta \vdash e' : \tau' \rightarrow \tau \)
* \( \Gamma; \delta \vdash E[e'] : \tau \)

**CASE:** \( E[e] = (\lambda x.e') \)

(23) \( \Gamma; \delta \vdash (\lambda x.e') e : \tau \)
(24) \( \Gamma; \delta \vdash e : \tau' \)
(25) \( \Gamma \vdash (S; e) \)
(26) \( \Gamma; \delta \vdash e' : \tau' \)
* \( \Gamma; \delta \vdash E[e'] : \tau \)

**CASE:** \( E[e] = v \)

(27) \( \Gamma; \delta \vdash v : \tau' \)
(28) \( \Gamma \vdash (S; e) \)
(29) \( \Gamma; \delta \vdash e' : \tau' \)
* \( \Gamma; \delta \vdash E[e'] : \tau \)

**CASE:** \( E[e] = \text{iter}(e, e'', x, y, e''') \)

(30) \( \Gamma; \delta \vdash \text{iter}(e, e'', x, y, e''') : \tau \)
(31) \( \Gamma; \delta \vdash e : \tau' \)
(32) \( \Gamma \vdash (S; e) \)
(33) \( \Gamma; \delta \vdash e' : \tau' \)
* \( \Gamma; \delta \vdash E[e'] : \tau' \)

**CASE:** \( E[e] = \text{match} e \) with \( x :: x :: e \rightarrow e'' | [ ] \rightarrow e''' \)

(34) \( \Gamma; \delta \vdash \text{match} e \) with \( x :: x :: e \rightarrow e'' | [ ] \rightarrow e''' : \tau \)
(35) \( \Gamma; \delta \vdash e : \tau' \)
(36) \( \Gamma \vdash (S; e) \)
(37) \( \Gamma; \delta \vdash e' : \tau' \)
* \( \Gamma; \delta \vdash E[e'] : \tau' \)

2. **RNAME**

(H1) \( \Gamma \vdash (S; a) \)
(H2) \( \Gamma; \delta \vdash a : \tau \)
(H3) \( S; a \rightarrow v \)
(4) \( S(a) = (e, v, s) \)
(5) \( \cdot \mid \Gamma \vdash S \)
* \( \Gamma; \delta \vdash v : \tau \)

3. **RAPP**

(H1) \( \Gamma \vdash (S; (\lambda x.e) v) \)
(H2) \( \Gamma; \delta \vdash (\lambda x.e) v : \tau' \)
(H3) \( S; (\lambda x.e) v \rightarrow e(v/x) \)
(4) \( \Gamma; \delta \vdash \lambda x.e : \tau \rightarrow \tau' \)
(5) \( \Gamma; \delta \vdash v : \tau \)
(6) \( \Gamma, x : \tau; \delta \vdash e : \tau' \)
* \( \Gamma; \delta \vdash e(v/x) : \tau' \)

4. **ROP**

(H1) \( \Gamma \vdash (S; v \ op \ v') \)
(H2) \( \Gamma; \delta \vdash v \ op \ v' : \tau'' \)
(H3) \( S; v \ op \ v' \rightarrow v'' \)
(4) \( \text{op}: \tau \rightarrow \tau' \rightarrow \tau'' \)
(5) \( v'' = [v \ op \ v'] \)
* \( \Gamma; \delta \vdash v'' : \tau'' \)

by inv. of **APP** in (H2)
by inv. of **APP** in (H2)
by inv. of **ABS** in (4)
by Lemma 2 with (6, 5)

by inv. of **ROP** in (H3)
from (4, 5)
5. **RTRUE**

(H1) \( \Gamma \vdash (S; \text{true} \? e : e') \)
(H2) \( \Gamma; \delta \vdash \text{true} \? e : e' : \tau \)
(H3) \( S; \text{true} \? e : e' \rightarrow e \)
   * \( \Gamma; \delta \vdash e : \tau \) \hspace{1cm} \text{by inv. of RTRUE in (H2)}

6. **RFALSE**

(H1) \( \Gamma \vdash (S; \text{false} \? e : e') \)
(H2) \( \Gamma; \delta \vdash \text{false} \? e : e' : \tau \)
(H3) \( S; \text{false} \? e : e' \rightarrow e \)
   * \( \Gamma; \delta \vdash e : \tau \) \hspace{1cm} \text{by inv. of RFALSE in (H2)}

7. **RNIL**

(H1) \( \Gamma \vdash (S; \text{iter}([], v, x, y, e)) \)
(H2) \( \Gamma; \delta \vdash \text{iter}([], v, x, y, e) : \tau \)
(H3) \( S; \text{iter}([], v, x, y, e) \rightarrow v \)
   * \( \Gamma; \delta \vdash v : \tau \) \hspace{1cm} \text{by inv. of iter in (H2)}

8. **RITERATE**

(H1) \( \Gamma \vdash (S; \text{iter}(z::zs, v, x, y, e)) \)
(H2) \( S; \text{iter}(z::zs, v, x, y, e) \rightarrow \text{iter}(zs, e\{z/x\}\{v/y\}, x, y, e) \)
(H3) \( \Gamma; \delta \vdash \text{iter}(z::zs, v, x, y, e) : \tau' \)
   * \( \Gamma; \delta \vdash z :: \tau' \) \hspace{1cm} \text{by inv. of iter in (H3)}
   * \( \Gamma; \delta \vdash v : \tau' \) \hspace{1cm} \text{by inv. of iter in (H3)}
   * \( \Gamma; \vdash \tau' \) \hspace{1cm} \text{by inv. of op in (4)}
   * \( \Gamma; \delta \vdash {\Lambda} : \tau \) \hspace{1cm} \text{by inv. of iter in (3)}
   * \( \Gamma; \delta \vdash z :: \tau' \) \hspace{1cm} \text{by Lemma 2 with (6, 7)}
   * \( \Gamma; \delta \vdash e\{z/x\} : \tau' \) \hspace{1cm} \text{by Lemma 2 with (9, 5)}
   * \( \Gamma; \delta \vdash \text{iter}(zs, e\{z/x\}\{v/y\}, x, y, e) : \tau' \) \hspace{1cm} \text{by iter with (8, 10, 6)}

9. **REMPY**

(H1) \( \Gamma \vdash (S; \text{match}[] \text{ with } x::xs \rightarrow e | [] \rightarrow e') \)
(H2) \( \Gamma; \delta \vdash (\text{match}[] \text{ with } x::xs \rightarrow e | [] \rightarrow e') : \tau \)
(H3) \( S; (\text{match}[] \text{ with } x::xs \rightarrow e | [] \rightarrow e') \rightarrow e' \)
   * \( \Gamma; \delta \vdash e' : \tau \) \hspace{1cm} \text{by inv. of MATCH in (H2)}

10. **RMATCH**

(H1) \( \Gamma \vdash (S; \text{match}xs \text{ with } x::xs \rightarrow e | [] \rightarrow e') \)
(H2) \( \Gamma; \delta \vdash (\text{match}xs \text{ with } x::xs \rightarrow e | [] \rightarrow e') : \tau' \)
(H3) \( S; (\text{match}xs \text{ with } x::xs \rightarrow e | [] \rightarrow e') \rightarrow e\{x/x\}\{v/x\}\{xs/xs\} \)
   * \( \Gamma; \delta \vdash x :: \tau' \) \hspace{1cm} \text{by inv. of MATCH in (H2)}
   * \( \Gamma; \vdash v :: \tau \) \hspace{1cm} \text{by op and (4)}
   * \( \Gamma; \delta \vdash v :: \tau \) \hspace{1cm} \text{by op and (4)}
   * \( \Gamma; \vdash e :: \tau \) \hspace{1cm} \text{by Lemma 2 with (5, 6)}
   * \( \Gamma; \delta \vdash e\{v/x\} \) \hspace{1cm} \text{by Lemma 2 with (8, 7)}
   * \( \Gamma; \delta \vdash e\{v/x\}\{xs/xs\} \)

**Corollary 1 (Equal Domains).** For all typing environments \( \Gamma \) and \( \Gamma' \), and states \( S \), if \( \Gamma \vdash \Gamma' \vdash S \), then \( \text{dom}(\Gamma') = \text{dom}(S) \).

**Lemma 6 (Weakening on Well-typed States).** For all typing environments \( \Gamma \) and states \( S \) such that \( a \notin \text{dom}(\Gamma) \) and \( \cdot \vdash S \) and \( \Gamma; \delta \vdash e :: \tau \),
1. If \( a :: \text{def}_a(\tau) \) then \( \cdot \vdash \Gamma; \alpha :: \text{def}_a(\tau) \rightarrow S, a \rightarrow (e, \square, [\square]) \).
2. If \( a :: \text{var}_a(\tau) \) then \( \cdot \vdash \Gamma; \alpha :: \text{var}_a(\tau) \rightarrow S, a \rightarrow (e, \square, [\square]) \).

**Proof.** We prove for each case based on the well-typed states, and well-formed queue definitions.

1. **Definition Name**

(H1) \( \cdot \vdash S \)
(H2) \( \Gamma; \delta \vdash e :: \tau \)
(H3) \( a \rightarrow (e, \square, [\square]) \)
(H4) \( a \notin \text{dom}(\Gamma) \)
(H5) \( \Gamma \vdash \cdot \)
(H6) \( \Gamma; a :: \text{def}_a(\tau) \vdash \cdot \)
(H7) \( S, a \rightarrow (e, \square, [\square]); \Gamma \vdash \cdot \) \hspace{1cm} \text{by Definition 9 in (H1)}
(H8) \( \Gamma; a :: \text{def}_a(\tau) \vdash \cdot \)
(H9) \( S, a \rightarrow (e, \square, [\square]); \Gamma \vdash \cdot \) \hspace{1cm} \text{by Definition 9 with (H4)}
(H10) \( \Gamma \vdash \cdot \) \hspace{1cm} \text{by Definition 7}
2. Variable Name

(H1) \( \Gamma \vdash S \)
(H2) \( \Gamma; \delta \vdash e : \tau \)
(H3) \( a \mapsto (e, \Box, []) \)
(H4) \( a \notin \text{dom}(\Gamma) \)
(H5) \( \Gamma \vdash \cdot \)
(H6) \( \Gamma, a : \var{\tau} \vdash \cdot \)
(H7) \( S, a \mapsto (e, \Box, []) ; \Gamma \vdash [] \)
(H8) \( S, a \mapsto (e, \Box, []) ; \Gamma \vdash [a] \)
(H9) \( \lnot \emptyset = \emptyset \)
(H10) \( \Gamma \vdash S \subseteq \text{dom}(S) \)
(H11) \( \Gamma \vdash \cdot \)
(H12) \( \Gamma \vdash a : \text{def}(\tau) \vdash \cdot \)

Lemma 7. For all typing environments \( \Gamma \) and states \( S \) such that \( \cdot \vdash S \) then, for all \( b \in \cup \{a\} \) \( \Rightarrow a \in \cap \{a\} \).

Proof. We prove by contradiction.

1. 
   (H1) \( \cup \{a\} \cap \cap \{a\} = \emptyset \)
   (H2) \( \exists b \in \cup \{a\} \wedge b \in \cap \{a\} \)
   (H3) \( a \notin \cap \{a\} \)
   (H4) \( a \in \cap \{a\} \)
   (H5) \( a \notin \cap \{a\} \)

   Contradiction (H1, 5)

Lemma 8 (Inversion). For all typing environments \( \Gamma, \Gamma' \) and states \( S, S' \) such that \( \Gamma \vdash \cdot \vdash S, a \mapsto (e, v, s) \) then, there is a typing environment \( \Gamma_1, \Gamma'_1 \) and a state \( S_1, S'_1 \) such that \( \Gamma = \Gamma_1, \Gamma'_1, S = S_1, S'_1, \) and \( \cup \{a\} = \text{dom}(S'_1) = \text{dom}(\Gamma'_1) \) and \( \Gamma_1 \vdash S_1, a \mapsto (e, v, s) \).

Proof. Follows directly from the definition of well-typed states (Definition 9) and Lemma 7.

Theorem 1 (Progress of Programs). For all program configurations \( (S; P ; L) \) and \( \forall a \in \text{dom}(S) . \ (S(a) = (e, \Box, s) \Rightarrow \Rightarrow a::L = [a]) \), then there is a program configuration \( (S'; P' ; L') \) such that \( (S; P ; L) \rightarrow (S'; P' ; L') \) and \( \forall a \in \text{dom}(S') . \ (S'(a) = (e, \Box, s) \Rightarrow \Rightarrow L' = [a]) \).

Proof. By induction on the length of the derivation \( \Gamma \vdash P \) we prove for the cases where \( L = [] \) and \( L \neq [] \).

1. \( a::L \neq [] \)
   (H1) \( \Gamma \vdash (S; P ; a::L) \)
   (H2) \( \forall a \in \text{dom}(S) . \ (S(a) = (e, \Box, s) \Rightarrow \Rightarrow a::L = [a]) \)
   (H3) \( \Gamma \vdash S \)
   (H4) \( S = (S[a \mapsto (e, v', s)] ; P ; a::L) \rightarrow (S[a \mapsto (e, v', s)] ; P ; a::L) \)
   (H5) \( \forall a \in \text{dom}(S) . \ (S(a) = (e, \Box, s) \Rightarrow \Rightarrow a::L = [a]) \)

2. \( a::L \neq [] \)
   (H1) \( \Gamma \vdash (S; P ; a::L) \)
   (H2) \( \forall a \in \text{dom}(S) . \ (S(a) = (e, \Box, s) \Rightarrow a::L = [a]) \)
   (H3) \( \Gamma \vdash S \)
   (H4) \( S; \Gamma \vdash a::L \)
   (H5) \( \Gamma \vdash (S; e) \)

Proof. By Definition 10 in (H1)

RQEXP with (4)

by Definition 9 in (H1)

by Definition 7 in (4)

by Definition 11 with (3, 5)

by Definition 9 in (3)
$S(a) = (e, v', s)$
e is a value $\vee (S; e) \rightarrow e'$ by Lemma 3 with (6, 5, 7)

**CASE**: $e$ is a value

$e = v$

* $(S; P; a(e); L) \rightarrow (S[a \mapsto (e, v, s)]; P; L)$

by RQVAL

(10)

**CASE**: $(S; e) \rightarrow e'$

* $(S; P; a(e); L) \rightarrow (S; P; a(e'); L)$

by RQSTEP with (10)

* $\forall a \in \text{dom}(S). (S(a) = (e, a, s) \Rightarrow a : L = [a])$

3. $L = []$

(H1) $\Gamma \vdash (S; P; [])$

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

(3) $\Gamma \vdash S$

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

**CASE**: DEF

$P = \text{def} a = e, P'$

(5) $\Gamma \vdash \text{def} a = e, P'$ by Definition 10 in (H1)

(6) $\text{def} a = e, P'$

(7) $a \notin \text{dom}(\Gamma)$ by inv. of DEF in (6)

(8) $a \notin \text{dom}(S)$ by (7, 4)

(9) $\psi(S, a) = []$ by def. of $\psi$ with (8)

(10) $S' = \text{subscribe}(S, a, e)$

* $(S; \text{def} a = e, P'; []) \rightarrow (S'[a \mapsto (e, \emptyset, []); P'; [a])$

by RDEF with (10, 9)

* $\forall a \in \text{dom}(S'). (S'(a) = (e, v, s))$

from (H2, 10)

(12)

$S'' = S'[a \mapsto (e, \emptyset, [])]$ by def. of $\psi$ with (17)

* $\forall a \in \text{dom}(S''). (S''(a) = (e, [a])$

from (12)

**CASE**: VAR

$P = \text{var} a = e, P'$

(15) $\Gamma \vdash \text{var} a = e, P'$ by Definition 10 in (H1)

(16) $\text{var} a = e, P'$

(17) $a \notin \text{dom}(\Gamma)$ by inv. of VAR in (15)

(18) $a \notin \text{dom}(S)$ by (16, 4)

(19) $\psi(S, a) = []$ by def. of $\psi$ and (17)

* $(S; \text{var} a = e, P'; []) \rightarrow (S[a \mapsto (e, \emptyset, []); P'; [a])$

by RVAR with (18)

* $\forall a \in \text{dom}(S). (S(a) = (e, v, s))$

from (H2)

(20)

* $\forall a \in \text{dom}(S[a \mapsto (e, \emptyset, [])]). (S(a) = (e, [a])$

from (20)

**CASE**: DO

$P = \text{do} e, P'$

(23) $\Gamma \vdash \text{do} e, P'$ by Definition 10 in (H1)

(24) $\text{do} e, P'$

(25) $\text{do} e : \text{Action}$ by inv. of DO in (23)

(26) $S; e \rightarrow e'$

* $(S; \text{do} e, P'; []) \rightarrow (S; \text{do} e', P'; [])$

by RDO with (25)

* $\forall a \in \text{dom}(S). (S(a) = (e, a, s) \Rightarrow L = [a])$

from (H2)

**CASE**: UVAR

$P = \text{var} a = e, P'$

(29) $\Gamma = \Gamma', a : \text{var}_s(\tau)$ by Definition 10 in (H1)

(30) $\text{var}_s(\tau) \vdash \text{var} a = e, P'$

(31) $a \in \text{dom}(\Gamma', a : \text{var}_s(\tau))$

(32) $a \in \text{dom}(S)$

* $(S; \text{var} a = e, P'; []) \rightarrow (S[a \mapsto (e, \emptyset, s); P'; [a])$

by RVAR with (33)

(33)

$S' = S[a \mapsto (e, \emptyset, s)]$ by def. of $\psi$ and (30)

* $\forall a \in \text{dom}(S'). (S'(a) = (e, s) \Rightarrow [a] = [a])$

from (H2, 33)

**CASE**: UVART

$P = \text{var} a = e, P'$

(36) $\Gamma = \Gamma', a : \text{var}_s(\tau)$ by Definition 10 in (H1)

(37) $\text{var}_s(\tau) \vdash \text{var} a = e, P'$

(38) $a \in \text{dom}(\Gamma', a : \text{var}_s(\tau))$

(39) $a \in \text{dom}(S)$

* $(S; \text{var} a = e, P'; []) \rightarrow (S[a \mapsto (e, \emptyset, s); P'; [a])$

by RVAR with (38)

(40)

$S' = S[a \mapsto (e, \emptyset, s)]$ by def. of $\psi$ and (37)

* $\forall a \in \text{dom}(S'). (S'(a) = (e, s) \Rightarrow [a] = [a])$

from (H2, 40)
\textbf{CASE: UDEF}

\[ \mathcal{P} = \text{def } a = e, \mathcal{P}' \]
\[ \Gamma = \Gamma', a : \text{def}_s(\tau) \]
\[ \Gamma', a : \text{def}_s(\tau) \vdash \text{def } a = e, \mathcal{P}' \]
\[ a \in \text{dom}(\Gamma'), a : \text{def}_s(\tau) \]
\[ a \in \text{dom}(\mathcal{S}) \]
\[ \psi(S, a) = s \]
\[ \mathcal{S}' = \text{subscribe}(S, a, e) \]
\[ \forall a \in \text{dom}(\mathcal{S}'). (\mathcal{S}'(a) = (e, v, s)) \]
\[ \mathcal{S}'' = \mathcal{S}'[a \mapsto (e, \Box, s)] \]
\[ \forall a \in \text{dom}(\mathcal{S}''). (\mathcal{S}''(a) = [e = a]) \]

\[ \text{by Definition 10 in (H1)} \]
\[ \text{by (43, 4)} \]
\[ \text{by def. of } \psi \text{ and (44)} \]
\[ \text{by RDEF with (45, 46)} \]
\[ \text{from (H2, 46)} \]
\[ \text{from (48)} \]

\textbf{CASE: VAREDEF}

\[ \mathcal{P} = \text{def } a = e, \mathcal{P}' \]
\[ \Gamma = \Gamma', a : \text{var}_s(\tau) \]
\[ \Gamma, a : \text{var}_s(\tau) \vdash \text{def } a = e, \mathcal{P}' \]
\[ a \in \text{dom}(\Gamma), a : \text{var}_s(\tau) \]
\[ a \in \text{dom}(\mathcal{S}) \]
\[ \psi(S, a) = s \]
\[ \mathcal{S}' = \text{subscribe}(S, a, e) \]
\[ \forall a \in \text{dom}(\mathcal{S}'). (\mathcal{S}'(a) = (e, v, s)) \]
\[ \mathcal{S}'' = \mathcal{S}'[a \mapsto (e, \Box, s)] \]
\[ \forall a \in \text{dom}(\mathcal{S}''). (\mathcal{S}''(a) = [e = a]) \]

\[ \text{by (51, 4)} \]
\[ \text{by def. of } \psi \text{ and (52)} \]
\[ \text{by RDEF with (53, 54)} \]
\[ \text{from (H2, 46)} \]
\[ \text{from (56)} \]

\textbf{Lemma 9.} For all states \( \mathcal{S} \) and typing environments \( \Gamma \) and \( \Gamma' \), if \( \Gamma, a : \text{var}_s(\tau) \vdash e, \mathcal{P} \) then \( \forall a \in \text{dom}(\mathcal{S}). \ m_s(a) \uparrow \).

\textbf{Proof.} By induction on the well-typed state derivation.

1. \( \Gamma, a : \text{var}_s(\tau) \vdash e, \mathcal{P} \).

   Trivially true for empty state

2. \( \Gamma, a : \text{var}_s(\tau) \vdash S, a \mapsto (e, a, s) \).

   \begin{align*}
   & (1) \quad \Gamma, a : \text{var}_s(\tau) \vdash \Gamma^' \vdash S \quad \text{by Definition 9} \\
   & (2) \quad \bigcup_{s}(s) \subseteq \text{dom}(\Gamma^') \quad \text{by Definition 9} \\
   & (3) \quad \forall b \in \text{dom}(\mathcal{S}). \ m_s(b) \uparrow \quad \text{by I.H. in (1)} \\
   & (4) \quad \text{dom}(\Gamma') = \text{dom}(\mathcal{S}) \quad \text{from Corollary 1 with (1)} \\
   & (5) \quad s \subseteq \text{dom}(\mathcal{S}) \quad \text{from (2, 4)} \\
   & (6) \quad m_s(a) = 1 + \sum_{b \in s} m_s(b) \quad \text{by Definition 12} \\
   & * \quad m_s(a) \uparrow \quad \text{from (6, 5, 3)}
   \end{align*}

3. \( \Gamma, a : \text{def}_s(\tau) \vdash S, a \mapsto (e, a, s) \).

   \begin{align*}
   & (1) \quad \Gamma, a : \text{def}_s(\tau) \vdash \Gamma^' \vdash S \quad \text{by Definition 9} \\
   & (2) \quad \bigcup_{s}(s) \subseteq \text{dom}(\Gamma^') \quad \text{by Definition 9} \\
   & (3) \quad \forall b \in \text{dom}(\mathcal{S}). \ m_s(b) \uparrow \quad \text{by I.H. in (1)} \\
   & (4) \quad \text{dom}(\Gamma') = \text{dom}(\mathcal{S}) \quad \text{from Corollary 1 with (1)} \\
   & (5) \quad s \subseteq \text{dom}(\mathcal{S}) \quad \text{from (2, 4)} \\
   & (6) \quad m_s(a) = 1 + \sum_{b \in s} m_s(b) \quad \text{by Definition 12} \\
   & * \quad m_s(a) \uparrow \quad \text{from (6, 5, 3)}
   \end{align*}

\textbf{Lemma 10 (Length Defined).} For all configurations \( (\mathcal{S}; \mathcal{P}; \mathcal{L}) \), and typing environments \( \Gamma \), if \( \Gamma \vdash (\mathcal{S}; \mathcal{P}; \mathcal{L}) \) then \( m_s(\mathcal{L}) \uparrow \).
Proof. Follows directly from Lemma 9 and the condition $\mathcal{L} \subseteq \text{dom}(\mathcal{S})$.

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

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

1. **RDEF**

   (H1) $\Gamma \vdash (\mathcal{S}; \text{def} \ a = e, \mathcal{P}; [])$

   (H2) $(\mathcal{S}; \text{def} \ a = e, \mathcal{P}; []) \rightarrow (\mathcal{S}'[a \mapsto (e, \Box, s)]; \mathcal{P}; [a])$

   (3) $\Gamma \vdash \text{def} \ a = e, \mathcal{P}$

   (4) $\text{dom}(\Gamma) = \text{dom}(\mathcal{S})$

   (5) $\mathcal{S} \Gamma \vdash []$

   (6) $a \notin \text{dom}(\Gamma)$

2. **CASE:** $a \notin \text{dom}(\Gamma)$

   (8) $a \notin \text{dom}(\mathcal{S})$

   (9) $\Gamma; \delta \vdash e : \tau$

   (10) $\Gamma, a : \text{def}_{\delta}(\tau) \vdash \mathcal{P}$

   (11) $\Gamma \vdash \text{def}_{\delta}(\tau) \vdash \mathcal{P}$

   (12) $\Gamma \vdash \text{def}_{\delta}(\tau) \vdash \mathcal{P}$

   (13) $\delta = \text{subscribe}(\mathcal{S}, a)$

3. **CASE:** $e$ and $e'$ have the same type

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

   (22) $a \notin \text{dom}(\delta)$

   (23) $\Gamma', a : \text{def}_{\delta}(\tau) \vdash \mathcal{P}$

   (24) $\Gamma_1 : \Gamma_1', a : \text{def}_{\delta}(\tau) \vdash \mathcal{S}_1, a \rightarrow (e', v', s')$

   (25) $\Gamma' = \Gamma_1$, $\Gamma_1'$

   (26) $\mathcal{S} = \mathcal{S}_1, a \rightarrow (e', v', s'), \mathcal{S}_1'$

   (27) $\mathcal{S}_1' = \text{subscribe}(\mathcal{S}, a)$

   (29) $a \notin \text{dom}(\Gamma_1)$

4. **CASE:** $a \in \text{dom}(\Gamma)$

   (21) $\mathcal{S}'$ and $\mathcal{S}'$ have different types

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

   $\mathcal{S}' = \text{subscribe}(\mathcal{S}, a)$

   $s = \psi(\mathcal{S}, a)$

   (38) $\Gamma ; \delta \vdash e : \tau$

   (39) $\Gamma' ; \delta \vdash e : \tau$

   (40) $a \notin \text{dom}(\Gamma)$

   (41) $\Gamma', a : \text{def}_{\delta}(\tau') \vdash \mathcal{P}$

   (42) $s = []$

   (43) $\mathcal{S}' = \mathcal{S}_1, a \rightarrow (e', o, s)$

   (44) $\Gamma' ; a : \text{def}_{\delta}(\tau') \vdash a \rightarrow (e', o, s)$

   by Definition 9 in (38) with (40)
43) \[ \Gamma \vdash S'' \] Isolation in (38) with (42)
44) \[ \Gamma', a : \text{def}_3(\tau) \vdash S'', a \rightarrow (e, \square, []) \] by Lemma 6 with (43, 39)
45) \[ S'[a \rightarrow (e, \square, s)] = S'', a \rightarrow (e, \square, []) \] by Definition 7 with (42)
46) \[ \Gamma', a : \text{def}_3(\tau) \vdash \langle S'[a \rightarrow (e, \square, s)]; P; [a] \rangle \] by Definition 10 with (44, 41, 45)

\[ \text{SCASE: VAR}\] Similar to the above sub-cases.

2. RVAR
Similar to RDEF cases.

3. RASSIGN

\[ (\text{H1}) \quad \Gamma \vdash (S[a \rightarrow (e', o, s)]; \text{do action} \{ a := e; \overline{\alpha'} := \overline{e'} \}; P; []) \]
\[ (\text{H2}) \quad (S[a \rightarrow (e', o, s)]; \text{do action} \{ a := e; \overline{\alpha'} := \overline{e'} \}; P; []) \]
\[ \quad \rightarrow (S[a \rightarrow (e', o, s)]; \text{do action} \{ \overline{\alpha'} := \overline{e'} \}; P; []) \]

(3) \[ \cdot \quad \Gamma \vdash S \] by Definition 10 in (H1)
(4) \[ \Gamma \vdash \text{do action} \{ a := e; \overline{\alpha'} := \overline{e'} \}, P \] by Definition 10 in (H1)
(5) \[ S; \Gamma \vdash [] \] by Definition 10 in (H1)
(6) \[ \Gamma; \delta \vdash \text{action} \{ a := e; \overline{\alpha'} := \overline{e'} \} : \text{Action} \] by inv. of DO in (4)
(7) \[ \Gamma \vdash P \] by inv. of ACTION in (6)
(8) \[ \Gamma; \delta \vdash a := e \] by inv. of ACTION in (6)
(9) \[ \Gamma; \delta \vdash \text{action} \{ \overline{\alpha'} := \overline{e'} \} : \text{Action} \] by DO with (8)
(10) \[ \Gamma \vdash \text{do action} \{ \overline{\alpha'} := \overline{e'} \}, P \] by DO with (9, 7)

4. RQEXP

\[ (\text{H1}) \quad \Gamma \vdash (S[a \rightarrow (e, o, s)]; P; a::\mathcal{L}) \]
\[ (\text{H2}) \quad (S[a \rightarrow (e, o, s)]; P; a::\mathcal{L}) \rightarrow (S[a \rightarrow (e, o, s)]; P; a(e)::\mathcal{L}) \]

(3) \[ \cdot \quad \Gamma \vdash S \] by Definition 10 in (H1)
(4) \[ \Gamma \vdash P \] by Definition 10 in (H1)
(5) \[ S[a \rightarrow (e, o, s)]; \Gamma \vdash a::\mathcal{L} \] by Definition 10 in (H1)
(6) \[ \Gamma; \delta \vdash e : \tau \] by Definition 9 with (3)
(7) \[ \Gamma \vdash (S[a \rightarrow (e, o, s)]; P; a(e)::\mathcal{L}) \] by inv. of Definition 10 with (3, 4, 7)

5. RQSTEP

\[ (\text{H1}) \quad \Gamma \vdash (S; P; a(e)::\mathcal{L}) \]
\[ (\text{H2}) \quad (S; P; a(e)::\mathcal{L}) \rightarrow (S; P; a(e')::\mathcal{L}) \]

(3) \[ \cdot \quad \Gamma \vdash S \] by Definition 10 in (H1)
(4) \[ \Gamma \vdash P \] by Definition 10 in (H1)
(5) \[ S; \Gamma \vdash a(e)::\mathcal{L} \] by Definition 10 in (H1)
(6) \[ \Gamma; a : \text{var}_3(\tau) \lor \Gamma = \Gamma', a : \text{def}_3(\tau) \] by Definition 7 in (5)
(7) \[ S; \Gamma \vdash a::\mathcal{L} \] by Definition 7 in (5)
(8) \[ S; e \rightarrow e' \] by RQSTEP in (H2)
(9) \[ \Gamma; \delta \vdash e' : \tau \] by Lemma 5 with (6, 8)
(10) \[ \Gamma \vdash (S; P; a(e')::\mathcal{L}) \] by Definition 7 with (9, 7)

6. RQVAL

\[ (\text{H1}) \quad \Gamma \vdash (S[a \rightarrow (e, o, s)]; P; a(v)::\mathcal{L}) \]
\[ (\text{H2}) \quad (S[a \rightarrow (e, o, s)]; P; a(v)::\mathcal{L}) \rightarrow (S[a \rightarrow (e, v, s)]; P; L_{\overline{o}s}) \]

(3) \[ \cdot \quad \Gamma \vdash S \] by Definition 10 in (H1)
(4) \[ \Gamma \vdash P \] by Definition 10 in (H1)
(5) \[ S; \Gamma \vdash a(v)::\mathcal{L} \] by Definition 10 in (H1)
(6) \[ \Gamma; \delta \vdash v : \tau \] by Definition 7 in (5)
(7) \[ S; \Gamma \vdash L_{\overline{o}s} \] by Definition 7 in (5)
(8) \[ a : \text{def}_3(\tau) \lor a : \text{var}_3(\tau) \] by Definition 9 in (3)

(7) \[ \text{CASE: a : def}_3(\tau) \]
\[ \Gamma = \Gamma_1, a : \text{def}_3(\tau), \Gamma_2 \]
\[ S = S_2, a \rightarrow (e, o, s), S_1 \]
\[ \Gamma_1 \vdash a : \text{def}_3(\tau), \Gamma_2 \vdash S_2, a \rightarrow (e, o, s) \] by Definition 9 in (3)
\begin{verbatim}(10)  \Gamma; \delta \vdash e : \tau          by Definition 9 in (9)
(11)  \Gamma, a : \text{def}_3(\pi) \vdash \Gamma_2 \vdash S_2     by Definition 9 in (9)
(12)  S_2, a \mapsto \{e, o, s\}; \Gamma_2 \vdash [a]     by Definition 9 in (9)
(13)  \Gamma \vdash \delta \subseteq \text{dom}(\Gamma_1)     by Definition 9 in (9)
(14)  \cup_{S_2}(s) \subseteq \text{dom}(\Gamma_2)     by Definition 9 in (9)
(15)  S_2, a \mapsto \{e, o, s\}; \Gamma_2 \vdash [a]     by Definition 7 and (12)
(16)  \Gamma_1; \delta \vdash v : \tau     from (6, 13)
(17)  \Gamma_1, a : \text{def}_3(\pi), \Gamma_2 \vdash S_2, a \mapsto \{e, o, s\}     by Definition 9 with (10, 16, 11, 15, 13, 14)
(18)  \Gamma_1, a : \text{def}_3(\pi), \Gamma_2 \vdash S_2, a \mapsto \{e, o, s\}     by Definition 9 with (17)
     from (8)
     + \Gamma \vdash (S[a \mapsto \{e, o, s\}]; P; \mathcal{L} \circ s)     by Definition 10 with (18, 4, 6)

(20) \text{CASE: } a : \text{var}_3(\tau)
     \Gamma = \Gamma_1, a : \text{var}_3(\pi), \Gamma_2
     \Sigma = S_2, a \mapsto \{e, o, s\}, \Sigma_1
     \Gamma_1, a : \text{var}_3(\pi) \vdash \Gamma_2 \vdash S_2
     \Sigma_2, a \mapsto \{e, o, s\}; \Gamma_2 \vdash [a]
     \Gamma_1, a : \text{var}_3(\pi), \Gamma_2 \vdash S_2, a \mapsto \{e, o, s\}, S_1
     \Sigma[a \mapsto \{e, o, s\}] = S_2, a \mapsto \{e, o, s\}, S_1
     \Gamma \vdash (S[a \mapsto \{e, o, s\}]; P; \mathcal{L} \circ s)
     \text{by Definition 10 with (28, 4, 6)}

7. RDO

(H1)  \Gamma \vdash (S; \text{do } e, P; [\varnothing])
     (S; \text{do } e, P; [\varnothing]) \rightarrow (S; \text{do } e, P; [\varnothing])     by Definition 10 in (H1)
     \Gamma \vdash S
     \text{by Definition 10 in (H1)}
     (S; \text{do } e, P; [\varnothing]) \rightarrow (S; \text{do } e, P; [\varnothing])     by Definition 10 in (H1)
     \Gamma \vdash \text{do } e, P
     \text{by Definition 10 in (H1)}
     \Gamma \vdash [\varnothing]     by Definition 10 in (H1)
     \Sigma; \Gamma \vdash [\varnothing]
     \text{by inv. of RDO in (H1)}
     \Sigma; e \rightarrow e'
     \text{by inv. of DO in (4)}
     \Gamma; \delta \vdash e : \text{Action}
     \text{by inv. of DO in (4)}
     \Gamma; \delta \vdash e' : \text{Action}
     \text{by Lemma 5 with (7, 6)}
     \Gamma \vdash \text{do } e', P
     \text{by DO with (9, 8)}
     \Gamma \vdash (S; \text{do } e', P; [\varnothing])     by Definition 10 with (3, 10, 5)

8. RSKIP

(H1)  \Gamma \vdash (S; \text{do action } \{\cdot\}, P; [\varnothing])
     (S; \text{do action } \{\cdot\}, P; [\varnothing]) \rightarrow (S; P; [\varnothing])     by Definition 10 in (H1)
     \Gamma \vdash \text{do action } \{\cdot\}, P
     \text{by inv. of DO in (3)}
     \Gamma \vdash P
     \text{by Definition 10 in (H1)}
     \Gamma \vdash [\varnothing]
     \text{by Definition 10 in (H1)}
     \Sigma; \Gamma \vdash [\varnothing]
     \text{by Definition 10 in (H1)}
     \Gamma \vdash (S; P; [\varnothing])     by Definition 10 with (5, 4, 6)

Lemma 1 (Queue Convergence). For all configurations \((S; P; \mathcal{L})\) and \((S'; P'; \mathcal{L}')\), and typing environments \(\Gamma\), if \(\Gamma \vdash (S; P; \mathcal{L})\) and \(\mathcal{L} \neq [\varnothing]\) then \((S; P; \mathcal{L}) \rightarrow \ast (S'; P'; \mathcal{L}')\) and \(m_\mathcal{L}(\mathcal{L}') < m_\mathcal{L}(\mathcal{L})\).

Proof. By induction on the structure of \((S; P; \mathcal{L})\).

1. RVAR

(H1)  \Gamma \vdash (S; \text{var } a = e, P; [\varnothing])
     (S; \text{var } a = e, P; [\varnothing]) \rightarrow (S'a \mapsto \{e, \Box, s\}; P; [a])     by RVAR
     * Trivially true because \(\mathcal{L} = [\varnothing]\)

2. RDEF

(H1)  \Gamma \vdash (S; \text{var } a = e, P; [\varnothing])
     (S; \text{var } a = e, P; [\varnothing]) \rightarrow (S'a \mapsto \{e, \Box, s\}; P; [a])     by RDEF
     * Trivially true because \(\mathcal{L} = [\varnothing]\)

3. RASSIGN

(H1)  \Gamma \vdash (S; \text{var } a = e, P; [\varnothing])
     (S; \text{var } a = e, P; [\varnothing]) \rightarrow (S'a \mapsto \{e, \Box, s\}; P; [a])     by RASSIGN
* Trivially true because $\mathcal{L} = []$

4. RQEXP

(H1) $(S[a \mapsto (e, o, s)]; P; a::\mathcal{L})$

(H2) $(S[a \mapsto (e, o, s)]; P; a::\mathcal{L}) \rightarrow (S[a \mapsto (e, o, s)]; P; a(e)::\mathcal{L})$

(2) $m_\#(a::\mathcal{L}) = 1 + m_\#(a) + m_\#(\mathcal{L})$

(3) $m_\#(a(e)::\mathcal{L}) = m_\#(a) + m_\#(\mathcal{L})$

(4) $m_\#(a(e)::\mathcal{L}) < m_\#(a::\mathcal{L})$

5. RQSTEP

(H1) $\Gamma \vdash (S[a \mapsto (e, o, s)]; P; a(e')::\mathcal{L})$

(H2) $S; e' \rightarrow^* v$

(3) $(S[a \mapsto (e, o, s)]; P; a(e')::\mathcal{L}) \rightarrow^* (S[a \mapsto (e, o, s)]; P; a(v)::\mathcal{L})$

(4) $(S[a \mapsto (e, o, s)]; P; a(v)::\mathcal{L}) \rightarrow (S[a \mapsto (e, v, s)]; P; \mathcal{L}@s)$

(5) $m_\#(a(e')::\mathcal{L}) = m_\#(a) + m_\#(\mathcal{L})$

(6) $m_\#(a) = 1 + \sum_{s \in s} m_\#(s) = 1 + m_\#(s)$

(7) $m_\#(a(e')::\mathcal{L}) = 1 + m_\#(s) + m_\#(\mathcal{L})$

(8) $m_\#(\mathcal{L}@s) = m_\#(\mathcal{L}) + m_\#(s)$

* $m_\#(a(e')::\mathcal{L}) < m_\#(\mathcal{L}@s)$

6. RQVAL

(H1) $\Gamma \vdash (S[a \mapsto (e, o, s)]; P; a(v)::\mathcal{L})$

(H2) $(S[a \mapsto (e, o, s)]; P; a(v)::\mathcal{L}) \rightarrow (S[a \mapsto (e, v, s)]; P; \mathcal{L}@s)$

(2) $m_\#(a(v)::\mathcal{L}) = m_\#(a) + m_\#(\mathcal{L})$

(3) $m_\#(a) = 1 + \sum_{s \in s} m_\#(s) = 1 + m_\#(s)$

(4) $m_\#(a(v)::\mathcal{L}) = 1 + m_\#(s) + m_\#(\mathcal{L})$

(5) $m_\#(\mathcal{L}@s) = m_\#(\mathcal{L}) + m_\#(s)$

* $m_\#(a(v)::\mathcal{L}) < m_\#(\mathcal{L}@s)$

7. RDO

(H1) $\Gamma \vdash (S; do e; P; [])$

(H2) $(S; do e; P; []) \rightarrow (S; do e'; P; [a])$

* Trivially true because $\mathcal{L} = []$

8. RSKIP

(H1) $\Gamma \vdash (S; do action \{ \_ \}; P; [])$

(H2) $(S; do action \{ \_ \}; P; []) \rightarrow (S; P; [])$

* Trivially true because $\mathcal{L} = []$