# Secure Information Flow in the Orc Concurrent Programming Language

Project Report

John A. Thywissen
The University of Texas at Austin
jthywiss@cs.utexas.edu

## Abstract

Orc is a concurrent, functional-like programming language. We extend Orc's type system with secure typing, to control the flow of information through programs according to a security policy. This policy is encoded in a lattice of labels that can be applied to values. The partial order of labels specifies the allowed information flows.

The impact of Orc's design and of concurrency in general are discussed. The data flows and control flows among Orc program expressions are analyzed, and the resulting rules are presented. Declassification (approved violations of the partial order) is possible in this design as a simple type assertion. Effects of Orc's trust of sites, and its current scheduling method are considered.

Twenty examples are demonstrated.

***Categories and Subject Descriptors*** D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls; D.3.3 [*Programming Languages*]: Language Constructs and Features—Security, Orc; D.2.0 [*Software Engineering*]: General—Protection mechanisms; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

***General Terms*** Security, Languages

***Keywords*** Secure information flow, security types, concurrency, declassification, Orc, stOrc

## 1. Introduction

Orc is a concurrent programming language that provides structured concurrency [Kitchin et al. 2009, Orc Research Team 2009]. In the 1970s, structured programming eliminated uncontrolled goto statements in favor of more abstract but constrained control flow structures, such as while loops. Orc advocates an analogous transition for concurrency. Rather than use low-level thread create, synchronize, and destroy primitives, Orc provides the programmer with four combinators to express how to orchestrate the concurrent evaluation of program expressions.

Orc is a (mostly) functional language, which provides basic constructs, like the combinators, and delegates all other work to sites. Sites can be thought of as remote function invocation (though most execute locally).

Final Project, 14 Dec 2009
C S 380S, *Theory and Practice of Secure Systems*
Fall 2009, Prof. Vitaly Shmatikov
Department of Computer Science, The University of Texas at Austin

Orc has an optional type-checked mode, which performs type inference using the Pierce-Turner Local Type Inference method [Pierce and Turner 2000].

A large class of envisioned Orc applications are long running processes, for example workflow procedures. These processes cannot be simply contained within a single user's login session because their duration far exceeds the duration of a typical session. Also, because Orc is focused on "orchestration" of services provided by its sites, Orc programs tend to draw data from disparate, often remote, sources. Therefore, it is more likely than a typical program that an Orc program will handle data from a mix of multiple domains, outside of the administratively simple environment of a single user's login session.

This raises the question of trustworthiness of Orc programs when they come into contact with sensitive data from various sources.

Herein, we analyze information flows in the standard Orc semantics, and construct rules for enforcing security properties using the Orc type checker.

## 2. Related Work

Briefly:

Smith [2006] provides a quick, readable introduction to secure information flow.

Sabelfeld and Myers [2003] provides an extensive survey of the secure information flow literature.

Secure information flow control by static analysis was first presented in Denning and Denning [1977], which provides a control flow graph based approach.

Type systems were demonstrated to be a good information flow control mechanism in Volpano et al. [1996].

Labels can be applied by various principals in support of their policies. Myers and Liskov [1997]'s decentralized labels approach enables information flow in a mixed policy domain environment.

Work up to this point was in purely sequential (single-threaded) models. Multi-threaded programs were addressed in Smith and Volpano [1998], which requires all loops' guards only use public data.

Myers [1999] presents JFlow, a secure information flow extension to Java. JFlow was apparently the first secure information flow extension for a widely-used language. Jif extends JFlow with, among other things, decentralized information flow.

Volpano and Smith [2000] argues that strict noninterference is not a workable policy.

Smith [2001] continues to work on the multi-threaded problem. It relaxes the condition requiring all loop guards to not use non-public data, but the proposed solution requires knowledge of timing of operations.

Pottier and Simonet [2003] describes a derivative of ML, with security typing.

Current state of the art includes quantitative information flow [Clarkson et al. 2009], where the information flow rates are quantified.

## 3. Questions under Study

Questions raised by the problem of secure information flow in Orc include:

- What are the various channels that information could flow among expressions in Orc (by design and covertly), under standard Orc semantics?
- To what extent can a secure information flow policy be enforced in standard Orc with appropriate security extensions?
- How does concurrency change the secure information flow problem or solution, if at all?
- Can a policy be implemented reasonably that is more practical than strict noninterference?
- Beyond the pure Orc semantic model, are there aspects of the implementation of the Orc compiler or runtime engine that affect information flows?

## 4. Contributions

Accordingly, this report provides:

- A characterization of information flow (all explicit and many implicit paths) in existing Orc. Some implicit paths are not considered, such as timing paths. Also, Orc sites are out of scope and site trustworthiness is assumed.
- A description of secure information flow extensions to the Orc language.

Also provided with this report is:

- stOrc, an extended version of the Orc compiler that labels types with a security label, and then type checks programs to enforce information flow policy.

## 5. Background: The Orc Programming Language

Orc is a language that focuses on concurrency. Orc's model of concurrent computations is one of *orchestration* of the interaction of *sites*. Sites perform all the computational work, and are treated similarly to a remote function call in conventional languages. Orc composes site calls with its four combinators: parallel, sequential, pruning, and otherwise. When an expression in an Orc program produces a result (called a publication), the result is propagated based on the semantics of the combinators surrounding it. Orc is similar to a dataflow language, in that publications become threads of control.

For example, the Orc expression $f() >x> g(x)$ causes the function $f$ to be executed, and then for *each* publication of $f$, the result value is bound to $x$ and $g(x)$ is evaluated.

***Sites*** When a site is called, it is optionally passed parameters and execution of that site call awaits a response from the site. Orc does not constrain the location of a site — it may be a remote call over, say, SOAP, or it may be a local method call in the caller's address space. The current Orc implementation runs in a Java environment, so site calls translate to method calls on subclasses of `Site`, but that's is nonessential to the language definition.

***Parallel*** The first combinator, parallel, written $\mid$ , has the expected meaning. Parallel executes the expressions on its left-hand side and right-hand side in parallel, and publishes all publications of both.

***Sequential*** The second combinator, sequential, written $\gg$ or $>x>$ , is a type of "fan-out" operator. Sequential executes the expression on its left-hand side, and then for *each* publication of that expression, the result value is bound to the variable specified ($x$) and the right-hand side is evaluated with that binding. Note that the right-hand side could be executed zero or more times, depending on how many times the left-hand side publishes. For example, $(1 \mid 2) >x> (x + 1)$ results in two publications, 2 and 3. If the right-hand expression does not refer to the bound variable, the combinator can be abbreviated $\gg$. Note that the right-hand expression is executed once for each publication of the left-hand expression, regardless of whether it refers to the bound variable. All publications of the right-hand expression are published by sequential.

***Pruning*** The third combinator, pruning, written $\ll$ or $<x<$ , is a type of choice operator. Pruning executes the expressions on both sides, but the left-hand side is executed with the variable specified ($x$) bound to a future. If an expression attempts to evaluate a future, evaluation suspends until the value of the future is available. When the right-hand side produces its *first* publication, that result becomes the value of the bound variable (replacing the future) and all further execution in the right-hand side is terminated (other results are said to be pruned). All publications of the left-hand expression are published by pruning.

***Otherwise*** The fourth combinator, otherwise, written `;` , is a type of "fall-back" handler. Normally, the left-hand expression is evaluated and its publications are published by otherwise. However, if all execution ceases in the left-hand expression without publishing any values, the right-hand expression is evaluated and its publications are published by otherwise.

***Stop*** `stop` is Orc's "sink" — publications that arrive at `stop` are discarded.

Unusually, the core Orc language performs no manipulation of data values. In fact, Orc only "understands" futures and closures — all other values are opaque to Orc and left to sites to interpret and manipulate.

## 6. Information Flow Channels in Orc

### 6.1 Explicit

Information flows in Orc that are intentional can be found by reviewing the language's semantics. These have been presented in several forms: tree semantics [Hoare et al. 2005], asynchronous operational semantics [Kitchin et al. 2006], timed operational and denotational semantics [Wehrman et al. 2008], and token semantics (appendix A).

***Data Flow*** Information flow within an Orc program is predominantly the data flow of the program. Data flow in Orc occurs in six ways:

1. Constant evaluation: A literal in the program text is evaluated.
2. Variable lookup: A value in the current environment is retrieved.
3. Publish: The current result is propagated to the enclosing language structure.
4. Variable binding: The current result is added to the current environment.

**Table 1.** Combinator data flows

| Combinator | In | Across | Out |
|---|---|---|---|
| Parallel $l \mid r$ | To both $l$ and $r$ | none | From both $l$ and $r$ |
| Sequential $l >x> r$ | To $l$ | $l$ to $r$ via $x$ | From $r$ |
| Pruning $l <x< r$ | To $l$ | $r$ to $l$ via $x$ | From $l$ |
| Otherwise $l \,;\, r$ | To both $l$ and possibly $r$ | none | From either $l$ or $r$ |

**Table 2.** Combinator control flows

| Combinator | In | Across | Out |
|---|---|---|---|
| Parallel $l \mid r$ | To both $l$ and $r$ | none | From both $l$ and $r$ |
| Sequential $l >x> r$ | To $l$ | $l$ to $r$ | From $r$ |
| Pruning $l <x< r$ | To $l$ | Across $r$ (in case of termination), and from $r$ to $l$ only if data flow | From $l$ |
| Otherwise $l \,;\, r$ | To $l$ | Possibly $l$ to $r$ (in case of termination) | From either $l$ or $r$ |

5. Site call and return: Parameter values are transmitted to a site and a return value is retrieved.
6. Through the four combinators, namely, parallel, sequential, pruning, and otherwise, detailed below.

The four Orc combinators propagate data as follows:

1. Parallel $l \mid r$: Evaluates both sides, $l$ and $r$ with the current environment. All publications of $l$ and $r$ become publications of the parallel combinator.
2. Sequential $l >x> r$: Evaluates $l$ with the current environment. For each publication of $l$, $x$ is bound to the result and $r$ is evaluated with the resulting environment. The publications of $r$ become publications of the sequential combinator.
3. Pruning $l <x< r$: Evaluates $r$ with the current environment, and evaluates $l$ with $x$ bound to a future. When $l$ attempts to evaluate the binding of $x$, $l$ will be suspended awaiting a value in place of the future. The *first* publication from $r$ replaces the future, all further activity in $r$ is pruned, and $l$ is resumed (if it was waiting). The publications of $l$ become the publications of the pruning combinator.
4. Otherwise $l \,;\, r$: Evaluates $l$ with the current environment, and if one or more publications are produced, those become the publications of the otherwise combinator. If all activity in $l$ ceases, then $r$ is evaluated with the current environment, and those publications become the publications of the otherwise combinator.

These data flows are summarized in table 1.

***Control Flow*** In addition to data flow, information flows in Orc programs in the form of control flow. Since core Orc is a functional language, control flow largely mirrors the data flow. This is summarized in table 2. However, there are several subtleties to three of the four combinators, which can be noted by comparing the two tables:

1. Sequential: The execution of $r$ indicates that $l$ published, even if $r$ does not depend on the value produced.
2. Pruning: Since $l$ is evaluated without waiting for $r$, there is no flow in the manner of sequential. If $l$ attempts to lookup $x$, it will wait on $r$, but the data flow and the control flow are identical in that case. However, the termination of all other activity in $r$ when it publishes is not captured in the data flow.
3. Otherwise: The execution of $r$ indicates that $l$ failed to publish.

Finally, site calls and returns are transfer of control flow to and from the callee site. The site chooses to return or not. For example, conditionals are implemented in Orc using the `if` site, which will only return if the condition evaluates as true.

### 6.2 Covert

The usual covert channels exist in Orc:

- Termination
- Timing
- Platform leaks

Additionally, while this isn't necessarily a proper *covert* channel, sites are trusted to not leak parameter values, results, or related information.

***Termination*** Termination or continuing operation of threads can leak information. For example, if a login process terminates (rather than spawning a number of subprocesses), that may indicate to an unauthorized observer that the presented login credentials were incorrect. Parallel systems provide an abundance of opportunities to observe termination.

Orc, in particular, with its combinators, uses thread termination as a routine aspect of its programs' control flow. In this way, Orc differs from conventional programming languages, and an information flow control in Orc must subsume the termination channel problem.

***Timing*** Timing channels continue to be vexing. Orc is similar to other parallel systems in this respect. As mentioned in the related work, current proposed solutions to the timing channel problem seem unworkable, and Orc does not seem to present any new opportunities for solving this problem.

Future work could analyze Orc's `Rtimer()` and `Ltimer()` sites for opportunities to reduce their use as a timing channel.

***Platform*** Platform leaks, such as power channels, electromagnetic channels, etc. are also present in Orc, since it runs on a conventional platform.

## 7. Enforcement of Secure Information Flow in Orc

Here, we present stORC, security typed Orc. stORC is an extension to Orc which enforces secure information flow by means of enhancing the Orc type system to enforce confidentiality constraints of security labeled data.

The existing Orc type system is supplemented with a new type, `SecurityLabledType`, which has to parameters: a regular Orc type and a security label. Security labels are values with a partial order that form a lattice. There is a security label designated as a default label, which all unlabeled Orc types are presumed to be labeled with.

Programmers can annotate any Orc type with a label, and use them in any place a type can be used. This includes ascribing types to literals, function or site call parameters, expressions, and so forth.

Orc's type inference carries the security labeled types through expressions — a type checked expression includes label checking, as detailed below.

### 7.1 Syntax

stOrc modifies Orc syntax [Orc Research Team 2009] very minimally – in just two areas:

Security labels are added:

*SecurityLabel* ::= "{" *Identifier* "}"
|

And these labels may be appended to any Orc type:

*Type* ::= "(" *Type,* * ")" *SecurityLabel*
| "`lambda`" *TypeFormals TypeListGroup*
":" *Type SecurityLabel*
| "`Top`" *SecurityLabel*
| "`Bot`" *SecurityLabel*
| "_" *SecurityLabel*
| *Identifier SecurityLabel*
| *Identifier* "`[`" *Type,* * "`]`" *SecurityLabel*

For example, an integer type that has a "secret" label is written `Integer{Secret}`.

### 7.2 Semantics

The operational semantics are completely unchanged, since Orc's typing is completely erased after type checking.

### 7.3 Typing rules

For any type $T$, the type is equal to itself labeled with the default label.

$$\frac{}{T = T\{\}} \text{ DEFAULT}$$

Applying another label to a labeled type results in a type labeled with the join of the labels.

$$\frac{}{T\{a\}\{b\} = T\{a \vee b\}} \text{ JOIN-LABELS}$$

For any two types with lablels $S\{a\}$ and $T\{b\}$, the type $S\{a\}$ is a subtype of $T\{b\}$ iff $S$ is a subtype of $T$ and $a \leq b$.

$$\frac{S \leq T \quad a \leq b}{S\{a\} \leq T\{b\}} \text{ LABELED-SUBTYPES}$$

To handle implicit (control) flows, Volpano et al. [1996] introduced phrase types and typing rules for command composition. This will not work in functional languages such as Orc. (See, for example security typing in ML [Pottier and Simonet 2003].)

In stOrc, every typing context is extended with a "control flow label", which can encodes the security level of the fact that evaluation is taking place in that context. The control flow label can be viewed as an analog of the *pc* variable in other secure information flow solutions.

Typing rules are as in standard Orc, with the following changes:

#### 7.3.1 Combinators

The combinators' typing rules become:

Parallel passes the incoming control flow label independently to its left side and right side. (The type, and therefore the label of the results, remains the join of the left and right sides.)

$$\frac{c; \Gamma \vdash l : T_l \quad c; \Gamma \vdash r : T_r}{c; \Gamma \vdash l \mid r : T_l \vee T_r} \text{ PARALLEL}$$

Sequential passes the label of the left side's type as the right side's control flow label, and joins the right side's published value type's label with the left side's label.

$$\frac{c; \Gamma \vdash l : T_l \quad c'; \Gamma, x : T_l \vdash r : T_r}{c; \Gamma \vdash l >x> r : T_r\{\text{label}(T_l)\}} \text{ SEQUENTIAL}$$
where $c' = c \vee \text{label}(T_l)$

Pruning passes the incoming control flow label independently to the left side and right side. (The type, and therefore the label of the results, remains the type of the left side.)

$$\frac{c; \Gamma, x : \tau_r \vdash l : T_l \quad c; \Gamma \vdash r : T_r}{c; \Gamma \vdash l <x< r : T_l} \text{ PRUNING}$$

Otherwise passes the label of its left side's type as the right side's control flow label. (The type, and therefore the label of the results, remains the join of the left and right sides.)

$$\frac{c; \Gamma \vdash l : T_l \quad c'; \Gamma \vdash r : T_r}{c; \Gamma \vdash l \,;\, r : T_l \vee T_r} \text{ OTHERWISE}$$
where $c' = c \vee \text{label}(T_l)$

#### 7.3.2 Site Calls

For a site call:

- Calling a site reveals control flow to this point, so all arguments are joined with the control flow label.
- For ease-of-use reasons, unlabeled formal parameters take label of the corresponding argument value, if labeled. This alleviates the need to relabel all existing routines' type declarations.
- Similarly, if site's result type is not labeled, it is labeled with the join of the arguments' labels.

$$\frac{c; \Gamma \vdash a_1 : T_1 \ldots}{c; \Gamma \vdash x(\overline{a}_n) : T\{c \vee c'\}} \text{ SITE-CALL-RESULT}$$
where $c' = \bigvee_n \text{label}(T_n)$

### 7.4 Comment on Soundness

Orc's type system is intended as an aid to developers, and is not a part of the core language. Unfortunately, the Orc type checker currently has no formal statement of the rules it enforces. Therefore, we are unable to present a soundness proof here.

The discussion above of data flow *vs.* control flow in Orc is our attempt at some justification for soundness of stOrc. However, this is not fully satisfactory, and it is future work to remedy this.

## 8. Beyond Strict Noninterference

One goal of this work was to investigate containment policies of a secure information flow other than strict noninterference. Strict noninterference quickly becomes an impractical policy in real programs.

Strict noninterference is defined as: all possible executions that have identical "low" inputs but possibly differing "high" inputs must have identical "low" outputs.

The canonical example of strict noninterference's impracticality is password checking. When the password-to-be-checked is compared against the correct-password, the results of the comparison would be precluded from being revealed to the supplier of the

password-to-be-checked. In fact, allowing a log in attempt to succeed or fail is a "leak". Many computations involving mixed levels are similar – they need access to secrets, but also need to communicate results to lower labeled contexts.

Thus, a need for declassification arises. This is a means of annotating a program fragment with an indication of trust that leaks of information to lower levels are permitted in specific instances.

In a security type system, practical declassification is very simple – it is just a type assertion (type cast). stORC uses Orc's type assertion mechanism in this manner. The declassification and password checker examples (sections 10.9 and 10.10) demonstrate this.

## 9. Implementation Effects

There are aspects of the implementation of the Orc compiler and runtime engine that affect secure information flows. Two significant ones are trust of sites and the current runtime's scheduling policy.

### 9.1 Trust of Sites

An Orc program's calls to sites, of course, reveal the arguments to the sites. In general, we trust sites to not disclose arguments or results beyond their specification. For sites that are not trusted, for example, `println`, the argument types can be annotated with appropriate labels. For instance, see the password checker example's `untrustedPrintln`.

Orc trusts the infrastructure that transfers arguments and results to and from sites. Most standard library sites run in the same Java virtual machine as the Orc program, so malicious sites could examine the Orc program's state and extract information not intended for that site. This is an implementation choice that can be modified if Orc programs often call untrusted local sites.

Additionally, remote sites have a proxy in the JVM that uses a communications method of that site's choice, providing opportunity for malicious or poorly-designed sites to leak arguments and results. Again, a partial solution is to declare sites with parameters labeled appropriately for untrusted sites.

### 9.2 Scheduling

The Orc language makes a very limited set of guarantees about scheduling of threads in a program, and is clear that the scheduling of multiple "ready" threads can be nondeterministic. However, the current reference implementation does, in fact, perform round robin scheduling of ready threads. This opens an opportunity for extremely high bit rate timing channels within a program. Since stORC currently disclaims handling timing channels, this is not a failure, but worthy of note.

## 10. Demonstrations

Here, we present a number of examples of small stORC programs. The examples are presented as Orc source code, with structured comments that indicate the program's type as inferred by the Orc type checker, and the program's output. (Most of these examples are automatically checked in stORC's regression test procedure.)

For these examples, we use the security label lattice shown in figure 1. The label {`A0`} is the default, "public" label, and is the label lattice's bottom element. The label {`F9`} is the top element.

Many examples use the Orc `val` syntax. `val x = d` applied to the expression `e` is syntactic sugar for `e <x< d`.

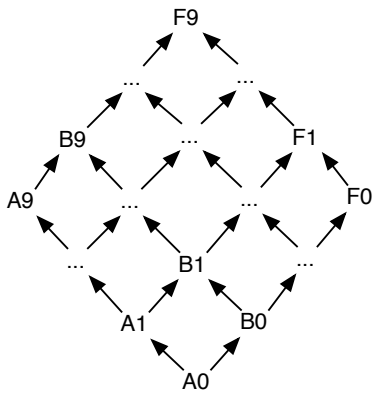


**Figure 1.** Security label lattice used in examples

### 10.1 Label use

Demonstrate the security label syntax in stORC

```
2 :: Integer{A2}

{-
TYPE:  Integer{A2}
OUTPUT:
2
-}
```

### 10.2 Multilevel add

Demonstrate a site call combining disparate argument labels

```
-- Add values of different labels,
-- check that output has correct inferred label.
-- An example of arrow type parameter/result label inference.

val public = 1 :: Integer
val secret1 = 4 :: Integer{B4}
val secret2 = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

public + secret1 + secret2

{-
TYPE:  Integer{B6}
OUTPUT:
11
-}
```

### 10.3 Parallel

Demonstrate the parallel combinator in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

low | high

{-
TYPE:  Integer{A6}
OUTPUT:
1
6
-}
```

### 10.4 Sequential

Demonstrate the sequential combinator in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

high >x> low

{-
TYPE:  Integer{A6}
OUTPUT:
1
-}
```

### 10.5 Pruning

Demonstrate the pruning combinator in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

low <x< high

{-
TYPE:  Integer
OUTPUT:
1
-}
```

### 10.6 Otherwise

Demonstrate the otherwise combinator in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

low ; high

{-
TYPE:  Integer{A6}
OUTPUT:
1
-}
```

### 10.7 Conditional simple

Demonstrate a simple control flow dependency in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

if(high > 2) >> "high > 2"

{-
TYPE:  String{A6}
OUTPUT:
"high > 2"
-}
```

### 10.8 Conditional otherwise

Demonstrate another control flow dependency in stOrc

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

(if(high > 2) >> "high > 2") ; "high <= 2"

{-
TYPE:  String{A6}
OUTPUT:
"high > 2"
-}
```

### 10.9 Declassification

Demonstrate declassification of a level B4 value to a level A0 (public) value

```
val low = 1 :: Integer
val mid = 4 :: Integer{4}
val high = 6 :: Integer{6}
val unused = 8 :: Integer{8}

def declassify(secret::Integer{6})::Integer{0} =
      secret:!:Integer{0}

declassify(mid) >x> x+100

{-
TYPE:  Integer{0}
OUTPUT:
104
-}
```

### 10.10 Password checker

Demonstrate both a trusted site, and an untrusted site.

`untrustedPrintln` must only be supplied public values as arguments.

checkPassword is allowed to publish a declassified Boolean.

```
val correctPassword = "secret" :: String{C5}

def untrustedPrintln(out::Top{A0})::Signal =
      println(out)

def checkPassword(String) :: Boolean
def checkPassword(enteredPassword) =
      (enteredPassword = correctPassword):!:Boolean{}

   untrustedPrintln("checkPassword(wrong)="
                    + checkPassword("wrong"))
>> untrustedPrintln("checkPassword(secret)="
                    + checkPassword("secret"))

{-
-- The following will not type check,
-- preventing a breach:

-- Try to reveal the secret
untrustedPrintln("correctPassword=" + correctPassword)
-}

{-
TYPE:  Top
OUTPUT:
checkPassword(wrong)=false
checkPassword(secret)=true
signal
-}
```

### 10.11 Adrian Quark examples

Quark [2009] gives the following nine examples of attempted breaches. All fail the STORC type checker.

Assume the following header for all examples:

```
val l = Ref[Boolean]()
val h = Ref[Boolean{C6}]()

-- The following 3 lines are a workaround for
-- type variable inference
type StoreType = Boolean
def (:=)(ref::Ref[StoreType], val::StoreType) =
      ref.write(val)
def (?)(ref::Ref[StoreType]) = ref.read()

h.write(true) >>
```

#### 10.11.1 Memory

```
l := h.read()
```

#### 10.11.2 Control flow

```
if h.read() then l := true else l := false
```

#### 10.11.3 Dynamic security failure

```
l := false >> if h.read() then l := true else signal
```

#### 10.11.4 Non-determinism

```
l := true | l := false | l := h.read()
```

#### 10.11.5 Compositionality

```
  ( h.write(true) >>
    Rtimer(10) >> if h.read() then l := true
                              else l := false
  )
|
  ( Rtimer(5) >> h.write(false)
  )
```

#### 10.11.6 Internal Timing

```
( Rtimer(50) >> l := true
| (if h.read() then Rtimer(100)
               else signal) >>
   l := false
)
```

#### 10.11.7 External Timing

```
l := true >>
(if h.read() then Rtimer(100)
             else signal) >>
l := false
```

#### 10.11.8 Synchronization

```
Semaphore(0) >s>
l := false >>
( s.acquire() >> l := true
| if(h.read()) >> s.release()
)
```

#### 10.11.9 Non-termination

```
def loop(x::Boolean)::Signal = if x then loop(x)
                                    else signal
h.write(true) >>

( Rtimer(50) >> l := true
| loop(h.read()) >> l := false
)
```

### 10.12 Timing leak

The following program demonstrates that STORC does not address timing channels. The program simply delays for a number of seconds equal to the value of the high variable.

```
val low = 1 :: Integer
val mid = 4 :: Integer{A4}
val high = 6 :: Integer{A6}
val unused = 9 :: Integer{F9}

val c = Clock()

Rtimer(high) >> stop ; c()

{-
TYPE:  Integer
OUTPUT:
6
-}
```

## 11. Conclusion

In a system with declassification, a malicious developer could simply declassify the secrets and write them on any desired output channel. Without declassification, but with termination and timing channels, these channels in real systems have high enough bit rates that a malicious developer could output substantial secrets quickly.

Therefore, stOrc's view of the developer is "trust, but verify". Design decisions for stOrc were driven by an assumption that the programmer is not malicious, but not perfect. Under this assumption, this work argues that secure information flow is a very practical addition to languages such as Orc.

Future work consists of engineering needed to make this a production-grade language feature to be integrated into the released version of Orc.

## 12. Future Work

- Add compiler support for user-declared security labels, along with a means of specifying the partial orders among labels. This involves syntax extensions and a more sophisticated `SecurityLabel` implementation, but no changes to other parts of stOrc.
- In addition to the current support, add integrity. The current framework supports this, with minor changes to the `SecurityLabel` class.
- A more explicit declassification annotation may be desirable. Type assertions are discouraged in Orc, so declassification should not appear to use it. It may also be desirable to syntactically differentiate declassification from "regular" type operations.
- Re-implement `SecurityLabledType` as a parameterized type. This and the next item bring up the question of whether labels should be values, and therefore should Orc's type system include dependent types. If not, then each label value could be treated as a type for easy use of the existing parametric polymorphism of Orc's type system.
- Handling labels as first class data, allowing them to be manipulated at run time, rather than type checked and erased, may seem desirable. However, this is a radical change that would eliminate the ability to completely statically check programs' safety.
- Add decentralized labels, as in Myers and Liskov [1997]. This requires the introduction of principals to the security system, and likely implies the previous item as well.
- Formalize the Orc type system, and prove soundness of stOrc's type system.
- Investigate reducing the `Rtimer`, `Clock` and `Ltimer` sites' potency for use as timing channels. Current language change proposals already include scoping for `Ltimer`; perhaps this scoping can be used to track the implicit control flow dependencies.
- Investigate quantitative information flow techniques to limit some of the covert channels' bit rates.

## Acknowledgments

Adrian Quark investigated secure information flow in untyped Orc[Quark 2009]. Use of his examples in section 10.11 is appreciated.

David Kitchin is the author of the Orc type checker, which has become the foundation of stOrc's implementation.

## References

Clarkson, M. R., Myers, A. C., and Schneider, F. B. 2009. Quantifying information flow with beliefs. *Journal of Computer Security 17,* 5, 655–701.

Denning, D. E. and Denning, P. J. 1977. Certification of programs for secure information flow. *Communications of the ACM 20,* 7, 504–513.

Hoare, T., Menzel, G., and Misra, J. 2005. A tree semantics of an orchestration language. In *Engineering Theories of Software Intensive Systems* (Marktoberdorf, Germany, 3–15 Aug 2004), M. Broy, D. Harel, T. Hoare, and J. Grünbauer, Eds. NATO Science Series II: Mathematics, Physics and Chemistry, vol. 195. Springer, 331–350.

Kitchin, D., Cook, W. R., and Misra, J. 2006. A language for task orchestration and its semantic properties. In *CONCUR 2006 – Concurrency Theory* (Bonn, Germany, 27–30 Aug 2006), C. Baier and H. Hermanns, Eds. Lecture Notes in Computer Science, vol. 4137. Springer, 477–491.

Kitchin, D., Quark, A., Cook, W. R., and Misra, J. 2009. The Orc programming language. In *Proceedings of FMOODS/FORTE 2009* (Lisbon, Portugal, 9–11 Jun 2009), D. Lee, A. Lopes, and A. Poetzsch-Heffter, Eds. Lecture Notes in Computer Science, vol. 5522. Springer, 1–25.

Myers, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (San Antonio, Tex., USA, 20–22 Jan 1999), A. Appel and A. Aiken, Eds. 228–241.

Myers, A. C. and Liskov, B. 1997. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (Saint Malo, France, 5-8 Oct 1997). 129–142.

Orc Research Team. 2009. *Orc User Guide*. Dept. of Computer Science, The Univ. of Texas at Austin. http://orc.csres.utexas.edu/.

Pierce, B. C. and Turner, D. N. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems 22,* 1 (Jan), 1–44.

Plotkin, G. D. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming 60–61*, 17–139.

Pottier, F. and Simonet, V. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems 25,* 1 (Jan), 117–158.

Quark, A. 2009. Orc secure information flow. Unpublished presentation.

Sabelfeld, A. and Myers, A. C. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21,* 1 (Jan), 5–19.

Smith, G. 2001. A new type system for secure information flow. In *Proceedings, 14th IEEE Computer Security Foundations Workshop* (Cape Breton, Novia Scotia, Canada, 11–13 June 2001). 115–125.

Smith, G. 2006. *Principles of Secure Information Flow Analysis*. Advances in Information Security, vol. 27. Springer, Chapter 13, 291–307.

Smith, G. and Volpano, D. 1998. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (San Diego, Calif., USA, 19–21 Jan 1998), D. B. MacQueen and L. Cardelli, Eds. 355–364.

Volpano, D. and Smith, G. 2000. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Boston, Mass., USA, 19–21 Jan 2000). 268–276.

Volpano, D., Smith, G., and Irvine, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4,* 2/3, 167–187.

Wehrman, I., Kitchin, D., Cook, W. R., and Misra, J. 2008. A timed semantics of Orc. *Theoretical Computer Science 402,* 2–3 (Aug), 234–248.

# A. Orc Token Semantics System

Orc's operational semantics are presented here for the reader's reference.

The system presented here is based on the structural operational semantics of Plotkin [2004], but instead of carrying environments on the left of a turnstile symbol, environments are carried in tokens. Also, instead of rewriting portions of a program's text to its evaluated value, the resulting value is carried in a token.

## A.1 Object Language Symbols

Phrases of object language of the Orc token semantics system include the following kinds of syntactic symbols:

***Abstract Lexeme*** Our abstract syntax of the source program is composed from the following symbols: literal values, identifiers, `stop`, $(,)$, $>$, $|$, $<$, $;$, $\stackrel{\text{def}}{=}$, and site declaration bodies.

For simplicity, assume all identifiers are unique – shadowing does not occur.

A "site declaration body" specifies the location of the interface to an Orc site. This is not elaborated further in this paper.

***Value*** A value, which is either a literal value, a closure, or a value returned by a site call. (In Orc, expressible and denotable values are the same set.)

***Environment*** A mapping from identifiers to values or $\bot$, which indicates a value that has not yet been computed.

***Tag Set*** An ordered set of opaque values (tags) which are used to identify groups of tokens. Sub-group membership is indicated by superset relationships.

***Token*** A structure used by the rewrite rules that carries state. The notation $\bullet^{v}_{\rho,\theta}$ represents a token with environment $\rho$, tag set $\theta$, and result $v$.

***Prototoken*** Another structure used by the rewrite rules that carries state. The notation $\odot_{\rho,\theta}$ represents a prototoken with environment $\rho$ and tag set $\theta$.

## A.2 Metavariables and Abstract Grammar

The OIL-derived AST definition can be instantiated into the following grammar, which is used by the rewrite rules in following sections:

***Expressions*** $d, d', e, e', l, l', r, r' \in \text{Exp} ::=$

- $c$ – literal value
- $x$ – variable
- `stop` – silent expression
- $x(\overline{a}_n)$ – site or function call
- $l >x> r$ – sequential
- $l \mid r$ – parallel
- $l <x< r$ – pruning
- $l \,;\, r$ – otherwise
- $x(\overline{x}_m) \stackrel{\text{def}}{=} d\ e$ – function definition and scope
- $x(\overline{x}_m) \stackrel{\text{def}}{=} s\ e$ – site declaration and scope
- $\overline{\bullet}\ e$ – expression ready for evaluation
- $e\ \overline{\bullet}$ – evaluation complete
- $x\overline{\bullet}(\overline{a}_n)$ – call in progress

where $\overline{\bullet}$ is a possibly empty sequence of:

- $\bullet^{v}_{\rho,\theta}$ – token with result
- $\bullet_{\rho,\theta}$ – token without result
- $\odot_{\rho,\theta}$ – prototoken

***Literal values (Constants)*** $c \in \text{Con}$ – integer literals, float literals, string literals, boolean literals, `signal`, and `null`.

***Variables*** $x \in \text{Var}$; $\overline{x}_m$ is a sequence of $m$ ($\geq 0$) variables, separated by commas

***Arguments*** $a_i \in \text{Con} \cup \text{Var}$; $\overline{a}_n$ is a sequence of $n$ ($\geq 0$) arguments, separated by commas

***Site declaration bodies*** $s \in \text{SiteDecBod}$, references to sites' locations in the external-to-Orc environment, not detailed here

***External values*** SiteRetVal. These external-to-Orc values are opaque values potentially returned by site calls. They are not detailed here.

***Closures*** Clo, tuples of the form $\langle \theta, m \rangle$ or $\langle s, m \rangle$

***Values*** $v, v' \in \text{Val} = \text{Con} \cup \text{Clo} \cup \text{SiteRetVal}$

***Environments*** $\rho, \rho' : \text{Var} \to \text{Val} \cup \{\bot\}$

***Cardinalities (of argument lists)*** $m, n \in \mathbb{N}$

***Tag sets*** $\theta, \theta' \in \mathcal{P}(\text{Tag})$

## A.3 Derivation Rules

These rules operate on the abstract grammar above.

### A.3.1 Values and stop

$$\frac{}{\bullet_{\rho,\theta}\ c \to c\ \bullet^{c}_{\rho,\theta}}\ \textsc{Literal}$$

$$\frac{x \in \rho \qquad \rho(x) \neq \bot}{\bullet_{\rho,\theta}\ x \to x\ \bullet^{\rho(x)}_{\rho,\theta}}\ \textsc{Variable}$$

$$\frac{}{\bullet_{\rho,\theta}\ \texttt{stop} \to \texttt{stop}}\ \textsc{Silent}$$

### A.3.2 Site Calls

$$\frac{\rho(x) = \langle s, m \rangle \qquad m = n \qquad a_{1\ldots n} \neq \bot}{\bullet_{\rho,\theta}\ x(\overline{a}_n) \xrightarrow{s!\langle \theta', \overline{a}_n \rangle} x\ \bullet_{\rho,\theta'}\ (\overline{a}_n)}\ \textsc{SiteCall-Issue}$$

where $\theta' = \text{newTag}(\theta)$

$$\frac{\text{result } \langle \theta, v \rangle \text{ ready} \qquad v \neq \textbf{null}}{x\ \bullet_{\rho,\theta}\ (\overline{a}_n) \xrightarrow{\theta?\langle v \rangle} x(\overline{a}_n)\ \bullet^{v}_{\rho,\theta'}}\ \textsc{SiteCall-Return}$$

where $\theta' = \text{popTag}(\theta)$

$$\frac{\text{result } \langle \theta, v \rangle \text{ ready} \qquad v = \textbf{null}}{x\ \bullet_{\rho,\theta}\ (\overline{a}_n) \xrightarrow{\theta?\langle v \rangle} x(\overline{a}_n)}\ \textsc{SiteCall-NullReturn}$$

### A.3.3 Function Calls

$$\frac{\rho(x) = \langle \theta_0, m \rangle \qquad m = n}{\bullet_{\rho,\theta}\ x(\overline{a}_n) \xrightarrow{\theta_0!\langle \theta', \overline{a}_n \rangle} x\ \odot_{\rho,\theta'}\ (\overline{a}_n)}\ \textsc{FuncCall-Issue}$$

where $\theta' = \text{newTag}(\theta)$

$$\frac{\text{result } \langle \theta', v \rangle \text{ ready}}{x\ \odot_{\rho,\theta}\ (\overline{a}_n) \xrightarrow{\theta'?\langle v \rangle} x\ \odot_{\rho,\theta}\ (\overline{a}_n)\ \bullet^{v}_{\rho,\theta'}}\ \textsc{FuncCall-Return}$$

where $\theta' = \text{popTag}(\theta)$

### A.3.4 Sequential Combinator

$$\frac{}{\bullet_{\rho,\theta}\ (l >x> r) \to (\bullet_{\rho,\theta}\ l) >x> r}\ \textsc{Sequential-Enter}$$

$$\frac{}{l\ \bullet^{v}_{\rho,\theta}\ >x> r \to l >x>\ \bullet_{\rho',\theta}\ r}\ \textsc{Sequential-PubL}$$

where $\rho' = \rho[x = v]$

$$\frac{}{l >x> (r \bullet^{v}_{\rho,\theta}) \to (l >x> r) \bullet^{v}_{\rho',\theta}} \text{ SEQUENTIAL-PUBR}$$
where $\rho' = \rho \setminus \{x\}$

### A.3.5 Parallel Combinator

$$\frac{}{\bullet_{\rho,\theta} (l \mid r) \to (\bullet_{\rho,\theta} l) \mid (\bullet_{\rho,\theta} r)} \text{ PARALLEL-ENTER}$$

$$\frac{}{(l \bullet^{v}_{\rho,\theta}) \mid r \to (l \mid r) \bullet^{v}_{\rho,\theta}} \text{ PARALLEL-PUBL}$$

$$\frac{}{l \mid (r \bullet^{v}_{\rho,\theta}) \to (l \mid r) \bullet^{v}_{\rho,\theta}} \text{ PARALLEL-PUBR}$$

### A.3.6 Pruning Combinator

$$\frac{}{\bullet_{\rho,\theta} (l <x< r) \to (\bullet_{\rho',\theta} l) <x< (\bullet_{\rho,\theta'} r)} \text{ PRUNING-ENTER}$$
where $\theta' = \text{newTag}(\theta) \quad \rho' = \rho[x = \bot_{\theta'}]$

$$\frac{}{(l \bullet^{v}_{\rho,\theta}) <x< r \to (l <x< r) \bullet^{v}_{\rho',\theta}} \text{ PRUNING-PUBL}$$
where $\rho' = \rho \setminus \{x\}$

$$\frac{}{l <x< (r \bullet^{v}_{\rho,\theta}) \to l' <x< r'} \text{ PRUNING-PUBR}$$
where $l' = \text{substEnvValue}(l, \bot_\theta, v) \quad r' = \text{eraseTokens}(\theta, r)$

### A.3.7 Otherwise Combinator

$$\frac{}{\bullet_{\rho,\theta} (l \,;\, r) \to (\bullet_{\rho,\theta'} l) \,;\, (\odot_{\rho,\theta'} r)} \text{ OTHERWISE-ENTER}$$
where $\theta' = \text{newTag}(\theta)$

$$\frac{\text{isLive}(\theta, l)}{l \,;\, \odot_{\rho,\theta} r \to l \,;\, \bullet_{\rho,\theta} r} \text{ OTHERWISE-NOPUB}$$

$$\frac{}{(l \bullet^{v}_{\rho,\theta}) \,;\, r \to (l \,;\, r') \bullet^{v}_{\rho,\theta'}} \text{ OTHERWISE-PUBL}$$
where $r' = \text{eraseTokens}(\theta, r) \quad \theta' = \text{popTag}(\theta)$

$$\frac{}{l \,;\, (r \bullet^{v}_{\rho,\theta}) \to (l \,;\, r) \bullet^{v}_{\rho,\theta'}} \text{ OTHERWISE-PUBR}$$
where $\theta' = \text{popTag}(\theta)$

### A.3.8 Site and Function Definitions

$$\frac{}{\bullet_{\rho,\theta} (x(\overline{x}_m) \stackrel{\text{def}}{=} s)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} s) \bullet_{\rho',\theta} e} \text{ DEF-SITEDEF}$$
where $\rho' = \rho[x = \langle s, m \rangle]$

$$\frac{}{\bullet_{\rho,\theta} (x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} \odot_{\rho,\theta'} d) \bullet_{\rho',\theta} e} \text{ DEF-FUNCDEF}$$
where $\theta' = \text{newTag}(\theta) \quad \rho' = \rho[x = \langle \theta', m \rangle]$

$$\frac{\langle \theta, \theta', \overline{a}_n \rangle \text{ sent}}{(x(\overline{x}_m) \stackrel{\text{def}}{=} \odot_{\rho,\theta} d)\ e \xrightarrow{\theta ? \langle \theta', \overline{a}_n \rangle} (x(\overline{x}_m) \stackrel{\text{def}}{=} \odot_{\rho,\theta} \bullet_{\rho',\theta'} d)\ e} \text{ DEF-ENTERBODY}$$
where $\rho' = \rho[x_1 = a_1, \ldots, x_m = a_n]$

$$\frac{}{(x(\overline{x}_m) \stackrel{\text{def}}{=} d \bullet^{v}_{\rho,\theta})\ e \xrightarrow{\theta ! \langle v \rangle} (x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e} \text{ DEF-PUBBODY}$$

$$\frac{}{((x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e \bullet^{v}_{\rho,\theta}) \to ((x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e) \bullet^{v}_{\rho',\theta}} \text{ DEF-PUBSCOPE}$$
where $\rho' = \rho \setminus \{x\}$

$$\frac{\neg\, \text{isLive}(\theta, d)}{x \odot_{\rho,\theta} (\overline{a}_n) \to x(\overline{a}_n)} \text{ FUNCCALL-CLEANDEAD}$$
where $d = x$'s def body

$$\frac{\neg\, \text{isLive}(\theta', e)}{(x(\overline{x}_m) \stackrel{\text{def}}{=} \odot_{\rho,\theta} d)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e} \text{ DEF-CLEANDEAD}$$
where $\theta' = \text{popTag}(\theta)$

### A.3.9 Congruence Rules

$$\frac{l \to l'}{l >x> r \to l' >x> r} \text{ SEQUENTIAL-CONGRUL}$$

$$\frac{r \to r'}{l >x> r \to l >x> r'} \text{ SEQUENTIAL-CONGRUR}$$

$$\frac{l \to l'}{l \mid r \to l' \mid r} \text{ PARALLEL-CONGRUL}$$

$$\frac{r \to r'}{l \mid r \to l \mid r'} \text{ PARALLEL-CONGRUR}$$

$$\frac{l \to l'}{l <x< r \to l' <x< r} \text{ PRUNING-CONGRUL}$$

$$\frac{r \to r'}{l <x< r \to l <x< r'} \text{ PRUNING-CONGRUR}$$

$$\frac{l \to l'}{l \,;\, r \to l' \,;\, r} \text{ OTHERWISE-CONGRL}$$

$$\frac{r \to r'}{l \,;\, r \to l \,;\, r'} \text{ OTHERWISE-CONGRR}$$

$$\frac{e \to e'}{(x(\overline{x}_m) \stackrel{\text{def}}{=} s)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} s)\ e'} \text{ DEF-SITECONGRU}$$

$$\frac{d \to d'}{(x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} d')\ e} \text{ DEF-CONGRUBODY}$$

$$\frac{e \to e'}{(x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e \to (x(\overline{x}_m) \stackrel{\text{def}}{=} d)\ e'} \text{ DEF-CONGRUSCOPE}$$

$$\frac{e \to e'}{\bullet_{\rho,\theta} e \to \bullet_{\rho,\theta} e'} \text{ TOKEN-CONGRU}$$

$$\frac{}{\bullet^{v}_{\rho,\theta} (\bullet^{v'}_{\rho',\theta'} e) \to \bullet^{v'}_{\rho',\theta'} (\bullet^{v}_{\rho,\theta} e)} \text{ TOKEN-COMMUTE}$$

$$\frac{e \to e'}{\odot_{\rho,\theta} e \to \odot_{\rho,\theta} e'} \text{ PROTOTOKEN-CONGRU}$$

$$\frac{}{\odot_{\rho,\theta} (\odot_{\rho',\theta'} e) \to \odot_{\rho',\theta'} (\odot_{\rho,\theta} e)} \text{ PROTOTOKEN-COMMUTE}$$

$$\frac{}{\bullet^{v}_{\rho,\theta} (\odot_{\rho',\theta'} e) \to \odot_{\rho',\theta'} (\bullet^{v}_{\rho,\theta} e)} \text{ TOKENPROTO-COMMUTE}$$

$$\frac{}{\bullet^{v}_{\rho,\theta} (e \bullet^{v'}_{\rho',\theta'}) \to (\bullet^{v}_{\rho,\theta} e) \bullet^{v'}_{\rho',\theta'}} \text{ TOKEN-ASSOC}$$

### A.4 Auxiliary Functions

$$\mathrm{newTag}(\theta) = \{\text{new tag value}\} \cup \theta$$

$\mathrm{popTag}(\theta) = \theta \setminus \theta_1$, where $\theta_1$ is the most recently added element

$$\mathrm{isLive}(\theta, e) \Leftrightarrow \mathrm{eraseTokens}(\theta, e) = e$$

$\mathrm{eraseTokens}(\theta, e)$ = walk lexemes in $e$ and erase all tokens $\bullet_{,\theta'}$ and all prototokens $\odot_{,\theta'}$, where $\theta' \supseteq \theta$. For all function calls $x \odot_{,\theta'} (\overline{a}_n)$ found in the expression $e$, apply $\mathrm{eraseTokens}$ to the function definition body.

$\mathrm{substEnvValue}(e, \bot_\theta, v)$ = walk lexemes in $e$, updating tokens' environments, changing any variable bound to $\bot_\theta$ to be bound to $v$.

### A.5 Program Launch and Results

In the following, $e$ is specifically the entire program.

$$\frac{}{e \xrightarrow{\text{run program}} \bullet_{\varnothing,\theta}\, e} \text{ PROGRAM-RUN}$$

where $\theta = \mathrm{newTag}(\varnothing)$

$$\frac{}{e\, \bullet^{v}_{\rho,\theta} \xrightarrow{\text{publish } v} e} \text{ PROGRAM-PUBLISH}$$

### A.6 Actions (Transition labels)

For labels on transitions, such as $l$ in $\xrightarrow{l}$:

Let $a!b$ mean "send", under the tag $a$, the value $b$. Sending can be modeled as adding the pair $(a, b)$ to a "sent" multiset.

Let $a?x$ mean "receive" a value sent under the tag $a$, into the variable $x$. Receiving can be modeled as selecting a pair $(a, b)$ in the "sent" multiset (matching on $a$), removing it from the multiset, and letting $x = b$ in the rest of the rule.

run program is an event from the environment to request start of program execution, not further specified here.

publish $v$ is an event sent to the environment to indicate a result value of program execution, not further specified here.