Static Type System for Information-Flow Control of JavaScript

Yonathan Volpin

September 2020

Abstract

We formalized a static information-flow security type system for a core of JavaScript. This was done in an intuitionistic, linear, and affine logic logical framework Celf. We defined a simple core of JavaScript and formalized its semantics into a language ("JavaScript Light" - JSL). We developed JSL as security typed with a formalized type-system. The type-system enforces progress sensitive noninterference - a flavor of noninterference that is compositional.

Keywords: JavaScript, Information-Flow Security, Logical Framework

Supervisors: Carsten Schürmann and Willard Rafnsson

Contents

Part I
Opening

1 Introduction

In the introduction, we briefly motivate our work, and give a high-level summary of our aproach. A further explanation of those terms, which is necessary in order to understand our solution, will be given in the ”Background” section.

Protecting digital information is becoming increasingly important. In many organizations the information is one of the most valuable assets and information security is a priority. Information security refers to the action of keeping information secure from unauthorized observation. In this thesis we address the problem of information security in JavaScript. We propose an enforcement of information flow control (IFC). We have developed a static enforcement of information security that will eliminate information leakage through progress of a JavaScript program execution. This improves on existing IFC enforcements that may have information leakage through progress of a program execution.

1.1 Information Flow Control

Information flow is the transfer of information between input and output of a program. There are programs that, due to bad implementation, have undesired flows, in which they will leak information to unintended observers. Information flow control (IFC) is the control of information flows in a manner that avoids such undesired flows. For example, assume a simple hierarchical lattice of the levels ”low” (L) and ”high” (H). Under that hierarchy, we have a set of programs that have all their information labelled with either L or H. In addition, we have a set of observers also labelled with L or H that only can observe the program’s memory which has a hierarchically equal or label than that of the observer. IFC is achieved by assuring that an observer has no ability to distinguish between runs of a program that has been provided (in relation to the observer) unauthorised input. This indistinguishability property is called ”non-interference” (NI). Implicit flow observation means that the observer has the ability to learn about any condition based on observing its clause effects. Therefore, in addition to explicit flows, also implicit flow is avoided (Sabelfeld et al, 2003). Note that NI (in the form of non progress sensitive and non termination sensitive, see background section for details) only assumes that the program is being observed command by command, and is not concerned with the observer’s ability to know anything about the progress of the program. A stronger flavour of NI is the assumption that the observer could learn about the progress of a program by observation of the output, this flavour is called ”progress sensitive non-interference” (PSNI) (Askarov et al, 2009). For the formal details of these definitions see the ”Background” section.

1.2 JavaScript

JavaScript (JS) is the most commonly used programming language today (2020 Developer Survey, 2020) due to its simple, multi-paradigm, and dynamically typed features. Further popularized by the web, JS is now ubiquitous, used for implementing a large variety of software, including client-side web applications, server-side code, desktop applications, mobile apps and embedded software. JavaScript is following the ECMA standard specifications which assure that web pages are interpreted similarly by different browsers. All JS programs could be desugared into (translated into a more syntactically simpler form) the simpler Core JS which is still maintaining the ECMA standards compliance (Guha et al, 2010).

1.3 JavaScript's Security Issues

JS is an interpreted dynamic language, meaning that it runs its code piece by piece on run time without compilation(Resource for developers, 2020). JS is prone to Information flow leaks. Such information leaks might, intentionally or unintentionally, cause serious privacy issues and violate legal regulation (for example leaking private user input into an ad without user consent). This motivates the enforcement of information-flow security in a program (Hedin et al. 2014). JS includes a dynamic feature which allows the program to run code given to it run-time (such as the command "eval"). Consequentially, it is hard to enforce statically NI on JS (as the string that will be run is not available prior to the run). Therefore only dynamic run-time monitors for IFC enforcement have been introduced so far (Hedin et al. 2012).

1.4 Dynamic Runtime Monitors Shortcomings

Existing IFC enforcements for JavaScript merely enforce a weak property of IFC (non progress sensitive, non termination sensitive), Thus the following JS program will be accepted (please note that throughout this paper we will use JavaScript-like code as examples):

```
1 var lowVar = 1;
2 while (highVar) {skip;}
3 outL(lowVar);
```

Where outL is a method that outputs its argument to a low observant (hence the "low" in "outL"), "lowVar" is a low labelled variable and "highVar" is a high labelled variable. In this example, the observation of the presence of the low output by a L labelled observer would leak the value of the variable of "highVar". Because, if the variable "highVar" will be true, the program will never make the progress to the point it outputs the low variable (progress sensitive).

Furthermore, a dynamic enforcement is a runtime monitor. This means that the runtime monitor transforms the program, into one that "self-destructs" before doing something that would violate the IFC property. Such a tool does not guarantee any

IFC property about the original program, as it alters its behaviour to avoid information leaks. Therefore, the original program can not safely be used without a dynamic monitor. (Hedin et al, 2012)

1.5 Logical Frameworks and Celf

A logical framework is a Meta-Language for formalizing of logical deductive systems through type system derivation. LF and CLF are examples of logical frameworks.

LF is lambda calculus based Meta-Language which is used as a tool for defining and presenting logic (Harper et al, 1993) for proof-development environments. LF is following a judgement as types principle. CLF is an concurrent extension of LF which supports linear logic. Developed by (Schack-Nielsen et al, 2008), Celf is an implementation of CLF, it is used for displaying logical traces of a deductive system. Celf is a metalanguage used for experimenting with deductive systems, specially applicable to programming language theory and logic . Celf gives the capability of defining and implementing type systems as deductive logical rules, and assists in evaluating and formalizing proofs of the programs well typed properties. Celf works with linear logic and is capable of defining programs with a heap and allowing them to have a state and side effects. And therefore it can be used for creating languages that could construct and handle objects in a manner similar to Core JavaScript.

2 Main Ideas

Our main idea in this thesis is to demonstrate a JavaScript - Core's IFC enforcement by a static type system (enabling PSNI enforcement).

We developed a type system which enforces PSNI. The enforcement is abstractly the following: assume that "p" is any program which can run on our JavaScript-like language and "g" defines which information flows are permitted in p.The predicate welltyped(g, p) will return true if p type-checks under g (i.e. if there is such a g which could take p as it's proposition). For example, in the following program, there is no satisfactory g for which welltyped(g, p) holds.

```
1 var lowVar = 0; // low variable
2 var highVar = true; // high variable
3 if (highVar) lowVar = 1; // high assignment inside a clause of a
4 //high condition, NI violation due to implicit flow leak.
```

There is a low label assignment inside the clause of a high label conditioned "if" command, and thus the value of low will implicitly tell what the value of high is.

This could be fixed by moving the low assignment outside of the clause of the high guard, and thus the following program will be accepted by welltyped(g, p):

```
1 var lowVar = 0;
2 var highVar = true;
3 if (highVar) skip;
```

```
lowVar = 1; // low assignment after the if command is allowed
```

Derivation examples as such could be find in the ”Implementation” section.
A less intuitive example could be the following:

```
var username = inL();  //low input
var password = inH();  // high input
var account = new Account(username, password); // high object
while(!(account.checkLogin())) { //high condition
    account.invokeMessage();
    password = inH();
    account = new Account(username, password);
}
outL(username); // low output, for example : tell 3rd party that
// a given account tried to log-in, which is a PSNI violation.
outH(account); // high output, example: successful log in.
```

In this program, there are a low input username and a high input password. These 2 inputs are used in order to create a high account object. The program will then run a while loop which will check whether the account can log in with this password. In case it doesn't, the loop will invoke a message telling the user to try typing its password again, and thus updating the password and account variables. The condition of the while loop command is high (as it is a method in a high object which contains the secret password) and thus low outputs after this loop violates PSNI properties (for more details see ”PSNI” section below) so the following low output of username is a violation.

This program will thus not type check in our system.
A simple fix could be to low-output the username **before** the the loop. As follows:

```
var username = inL();  //low input
var password = inH();  // high input
var account = new Account(username, password); // high object
outL(username); // low output, e.g. tell a 3rd party that given
// account tempted to log-in, which now is not a PSNI violation.
while(!(account.checkLogin())) { //high condition
    account.invokeMessage();
    password = inH();
    account = new Account(username, password);
}
outH(account); // high output, e.g. successful log in.
```

And thus the program now doesn't violate PSNI and will type check.
Our goals in this project are:

- Create a prototype JavaScript Core like language (JSL) with a static type system which enforces IFC.
- Prove the type system soundness in regards to IFC.

2.1 Contribution

Using Celf, we have developed a Core JavaScript -like (JSL) language which allows us to use a variate of features (such as states, reference, objects, loops and exceptions) in a logical traceable and sound manner. Such approach has allowed us to develop a static type system with Celf's experimental tools. Static enforcement of IFC has been developed through the static type system. Such enforcement will have the benefits of a stronger IFC flavor, namely progress sensitive non-interference (PSNI), and the possibility of proving IFC implementation in advance prior to the run of the program. Therefore, in the scope of this paper, we will only consider the subset of JavaScript programs that can be formalized in our language JSL.

Part II
Background

3 JavaScript

3.1 Features

As JavaScript has been developed through the years with the client side developer needs in mind, variety of features have been implemented: Document Object Model (DOM) manipulation, Functions as First-Class Objects, Java like features and more. We will include the semantics of many of those features in our language to demonstrate our IFC securing type system.

3.2 Semantics

Since our goal is to create a sound type system, we had to construct semantics for our language. In order to do that we had to comply with the ECMA standards and to choose a model of operational semantics. Our operational semantics should be a sufficient inclusion of the JS features in a way that could be use full for our purpose of IFC enforcing static type system .

3.2.1 Standards

ECMAScript is a standard specifications for a scripting language (ECMAScript language specification, 1999). Despite JS following these specification, those alone are not specific enough to follow for verifiable formal semantics, and thus is insufficiently clear for developing a sound type system. This were merely used as a guide line of semantics in a loose way. We had to follow those specifications, but we also required a solid and precise model.

3.2.2 Desugaring

Finally we decided to follow the core of JS as presented by (Guha et al, 2010). This approach is presenting a core version of JavaScript which is, in contrast to the other version, traceable and sound, and thus fit for a sound IFC enforcing type system. We used a subset of the Core JS inspired by (Hedin et al, 2014) in a way which we found sufficient the demonstrate the ability to enforce IFC statically.

4 Information-Flow Security

Information Flow Control is a mechanism which enforces policies on a process of transferring information. The aim of IFC is to prevent an unauthorized flow of secret information to unintended outputs (thus an unintended flow) through leaks (Sabelfeld and Myers, 2003). The definition of a leak is a flow which is unintended. In the scope

of this project, leak will refer to a flow which stores high secure (secret) information in a low secure (public) memory (the non- interference property). In this paper, for simplicity, the implementation of the security hierarchy will include only low("L") and high("H") in the lattice. This should be sufficient for proof of concept in a later work more levels could easily be added to the lattice.

4.1 Non-Interference

The property of non-interference (NI) intuitively means that there are no leaks in a program, thus a high input may not just only be stored in a low(er) variable, but also not make any distinguishable change in it. A program satisfies the non-interference property if its low security memory do not depend on the high security inputs.

Thus a sound type system means that there is non-interference property for all its programs that are well typed.

4.2 Explicit/Implicit Flows

Explicit flows are direct assignment of high value to a low variable, f.ex:

```
lowVar = highVar;
```

Implicit flows are assignments under high labelled guard condition which reveal the guards value, f.ex:

```
if highVar {
    lowVar = true;
}
else lowVar = false;
```

One can track these flows in a type system using a program counter (pc) level. This example will leak highVar based on lowVar. In our case, the way this will be avoided is by tracking these flows in a type system, giving every expression and command an addition pc level, which will keep count of the current context of the expression. If it will be run inside a high guard, the level of the pc will be raised to high as well, and low assignments will be forbidden. A "high guard" will be any conditional command, thus while loops, try catch statements and if then else clauses will be be considered as such and handled appropriately as will be shown in the type system implementation.

4.3 Termination Flows

As defined above, in order to ensure non-interference, programs output on low destination must be indistinguishable under different high inputs. So far we only regarded indistinguishably as equality, however there could be other ways to distinguish between outputs, for example the observation of termination and progress of the program.

Termination-Sensitive Non-Interference ("TSNI") is a flavor of IFC that takes into account that ability of the public to observe the termination of a program, this could only leak up 1 bit of information per program run. f.ex:

```
if (highVar == lowVar) {
    while(true) {
    skip;
    }
}
```

The termination of the following program will leak whether highVar == lowVar (Heidin et al. 2012).

4.4 Progress Flows

Progress-Sensitive Non-Interference ("PSNI") is a flavor of IFC that takes into account that ability of the public to observe the progress of a program (Heidin et al. 2012). This approach is more permissive than the TSNI as it assumes that the attacker can only observe non termination if an following expected low output is missing. The following shows that more just a bit of information could be leaked duo to progress observation: (Moore et al, 2012)

```
int i = 0;
while(true) {
    while(i == secret) {
        skip;
    }
    i++;
    outL(0);
}
```

secret is a confidential positive integer. The program will output zeros equal times to the value of secret. As shown by (Askarov et al, 2008) a program as this will leak a 32-bit integer secret in under 6 seconds.

4.5 Enforcements

Dynamic Enforcement Dynamic IFC enforcements come in shape of run-time monitors. Those are keeping track of the information flow in relation to it context command by command, they accept non safe programs and just stop the further execution of the program (without terminating) once a violating command appears. These monitors are useful for the user to assure that this specific run of the program does not leak sensitive information, however it doesn't guarantee anything about the original program itself. In addition, dynamic enforcements are useful in dynamic language where external code

can be added to the program on run time. Due to the feature of analysing code only on run-time, progress sensitive can not be assured. e.g. the following program:

```
1 if (highVar == 1) { outL(1); }
2 outL(0);
```

in this example, in case the high condition will be false, the run-time monitor will not check the body of the if clause and thus 0 will be low-output. However if the condition is true the low-output inside the if clause will violate implicit flow NI and thus the program will stop without any output. Thus the value of high will always leak based on the low-output in a dynamic run-time monitor. Thus a dynamic enforcement is only capable of enforcing termination insensitive, progress insensitive non-interference (TINI and PINI) (Heidin et al, 2012).

Static Enforcement On the other hand, static enforcements come in shape of type system. Static type system means that the expression syntax, operations semantics and type system are defined and checked before the run of a program in order to assure its safe running guarantee. And in the case of IFC it enables static analysis. Static analysis has the advantages of a stronger progress sensitive IFC enforcement (Heidin et al, 2012).

5 Linear Logic in Celf

Linear logic resources are consumable. Resources have to be used exactly once, this works well with programming languages (such as JS) as each command has to be executed once. In Celf, linear logic formulas are polarized into positive and negative classes, and a monad is preventing the interference between them.

$$\begin{array}{lllll} \textit{negative}: & A^-, B^- & ::= & P^- | \Pi x : T.A^- | A^+ \multimap \{\mathrm{B}^+\} \mid A^+ \text{-@} \{B^+\} \\ \textit{positive}: & A^+, B^+ & ::= & A^+ \otimes B^+ | 1 | !A^- | A^- | @A^- \end{array}$$

linear implication $\multimap$: consuming the left hand fact and producing the right hand fact, example : dollar $\multimap$ { apple } will mean that a dollar is consumed and an apple is produced.

Simultaneous Conjunction $\otimes$: Assume that an apple costs 1 dollar and 1 orange, the bind of a dollar and an orange in the transaction will be formalized with an $\otimes$. Example : dollar $\otimes$ orange $\multimap$ { apple } , will mean that a dollar and an orange are consumed and an apple in produced.

unrestricted modality ! : Assume the cost of an apple is 1 dollar, but also requires acquiring an orange (so the transaction will only consume the dollar but not the orange) the formalization of a resource that doesn't get consumed once used is by !, example: dollar $\otimes$!orange $\multimap$ apple will mean that a dollar will be consumed but but the orange wouldn't.

Universal Quantification $\Pi x : T$ lets assume that those dollar and orange must be of the same person, the $\Pi x : T$ could be used as a domain parameter. example : Πp :

$person.(dollar(p)(x)!orange(p) \multimap \{apple(p)\})$ thus all the facts are related to the specific person p.

@ means that a resource might or might not be consumed. -@ is consuming a resource that might or might not be consumed (Schack-Nielsen et al, 2008).

Part III
Contribution

6 JavaScript Light (JSL)

JavaScript has many variations, we decided to focus on Core JavaScript as it includes the relevant parts of securing information flow control. Our implementation will include exceptions, high order functions and object following the ECMA-262 standards. However dynamic code evaluation features will be omitted due to our prior to run analysis approach (as dynamic code evaluation is only presented during run-time, it can not be analysed prior to run (Heidin et al. 2012)). Our prototype is a JavaScript Core like language, "JavaScript Light" (JSL), that is amenable to construct a proof for the soundness of its programs. This toy-language was written in Celf, because of its unique ability to construct proofs of the type system and implement formal logic.
Our encoding of the language consists of expressions, commands, operational semantics and type system as explained:

- Expressions: expressions are defined by an abstract syntax of a language (I.S.O, 1999).

- Operational Semantics: This process of a language interpretation of an expression in order to produce another value as an expression .
  Operational semantics are operationally assigning formal meaning to expressions, in terms of how the expression computes, step-wise (Pfenning, 1992).

- Type System: Types are predefined attributes given to expressions and are allocated to types system rules. The types system defines and restricts the interaction between expression (Pfenning, 1992). This is done by enforcing relationship between expressions to be fitting with the type system rules by the compiler. This is often done in the pre-run to avoid run-time errors. In the case of IFC the type system is also used in order to prevent information leaks, as it will make sure that low level labeled variables will not receive high level values.

6.1 Expressions syntax

The Expressions syntax of our language is a subset of the non-strict semantics of ECMA-262 (v.5) standard. In the current version of our implementation, several of the expression of the standards are missing, especially those that are related to dynamic operations such as "eval". However our prototype should still be sufficient for a static analysis of the static subset of JS Core programs. The syntax is divided into levels. types, L-types, references,continuations, fields, objects, expressions and commands.

Distinguished as following :

$$
\begin{array}{rlll}
\text{Levels:} & l & ::= & H \mid L \\
\text{Types:} & T & ::= & 1 \mid \text{nat} \mid \text{bool} \mid l_1; \theta_1 \rightarrow^{l_2} \theta_2 \mid \text{fieldlist} \mid \text{objectable} \\
\text{L-types:} & \theta & ::= & T^l \\
 & & & \\
\text{Continuations:} & \kappa & & \\
\text{Properties:} & v & & \\
\text{Objects:} & o & ::= & \text{global} \mid n \\
 & & & \\
\text{Expressions:} & e & ::= x & \mid \overline{n} \mid \text{dec}(e) \\
 & & & \mid \text{true} \mid \text{false} \mid \neg e \\
 & & & \mid () \mid \bot \\
 & & & \mid e_1 = e_2 \\
 & & & \mid \lambda x.c \mid e_1 e_2 \\
 & & & \mid \mu f.c \\
 & & & \mid e_1.v := e_2 \mid e.v \\
 & & & \mid !o \mid \{\} \mid \text{new } e_1(e_2) \\
 & & & \\
\text{Commands:} & c & ::= & e \\
 & & & \mid \text{newproperty } v.c \\
 & & & \mid \text{var } e_1.v : \theta = o \mapsto e_2 \\
 & & & \mid \text{skip} \mid \text{return}(e) \mid c_1; c_2 \\
 & & & \mid \text{while}(e)\ \{c\} \mid \text{if}(e)\ \{c_1\}\ \text{else}\ \{c_2\} \\
 & & & \mid \text{try}\ \{c_1\ : \theta\}\ \text{catch}\ \{c_2\}\ \text{finally}\ \{c_3\} \\
 & & & \mid \text{throw}(\kappa, e)
\end{array}
$$

Levels are security levels used to label types for IFC purposes. Types and labeled types will be explained in the type system section. Types are denoted as T. 1 is the type of a unit. Nat is the type of a natural number. Bool is the type of a boolean. θ is a type labeled with a security level. $l_1; \theta_1 \rightarrow^{l_2} \theta_2$ is the type of a function, "fieldlist" is the type of the list of properties which are attached to an object. "objectable" is the type of objects.

Continuations, denoted as κ. These are used as the throw statement inside the try clause in a try-catch-finally command (see semantics section).

Properties are denoted as v and are the fields of objects.

Objects are denoted as o and are either the global object (global object is the root of all objects in the program. Meaning that all the objects in a program are properties of the global object) or any other code made object denoted n.

We will use unary natural numbers (Church, 1985), meaning that zero is defined as "z" and the rest of the numbers are defined by the successors of zero "s" in this way, every number is defined as the successor of the previous number (f.ex. "(s z)" is 1 and "(s(s z))" is 2 and so on). Natural numbers are denoted as "n". dec(e) is an expression which takes a natural number and returns it's value decreased by 1. true and false are boolean, $\neg$ is taking a boolean and returns its boolean opposite. () is a unit (empty object) and $\bot$ is undefined (empty field). = is comparison between 2 expression, λ x.c is a function which takes x as argument and runs command c. e1e2 is application of a

function e1 with input e2, μf.c is a recursive function which takes function f and runs command c. e1.v:=e2 is assignment of e1.v where e1 is an object and v is a property of that object, and e2 is a expression which will be assigned to that property. !o is a reference of a scope as an expression. {} is a derefernce of an object."new e1(e2)" is creating a new object, where expression e1 is a constructor and expression e2 is an argument to the constructor.

commands are denoted as c. every expression e is also a command. newproperty v.c is declaration of a new property v which will exist inside command c. var $e_1.v : \theta = o \mapsto e_2$ is the declaration of a new property v inside object e_1 of type θ assigned with object o which is mapping to expression e_2.

skip is the empty command which doesn't do anything.

return(e) is returning argument expression e as a result of its clause.

"$c_1; c_2$" for a sequential command, running the first command and adding the second command to the command stack.

"while" loop that takes a boolean expression and keeps running the command in loops as long as the expression is true,

The pattern matching of expression is through the operation of "if(e) $\{c_1\}$ else $\{c_2\}$". The evaluation of "if" is that it takes input a boolean ("e") and compares is to "true". If it matches, it will run the first command ("c") and if not it will run the second command.

"try" "catch" "finally" "throw" mechanism, runs the try clause, if a throw(κ, e) command is called than the catch clause will immediately be run with the throw expression variable as continuation argument, and run the finally clause regardless if throw was called or not.

6.2 Operational Semantics

first we should define the expressions syntax, this is done in Celf in a similar way to Twelf, where the "type" is the built in key word for creating a type category. And thus new syntax could be assigned to each category. Thus notice that a syntactic category is ": type".

```
level : type.
high : level.
low : level.
```

Thus "level : type" means that level is a syntatic category, and "high : level" means that high is type of the category level.

```
tau : type.
tp : type.

rec : tp.
one : tp.
nat : tp.
```

```
bool : tp.
prod : tp → tp → tp.
% add a case for classes, or maybe just a type for "objects".

=>' : level → level → tau → tau → tp. % with BL now
#' :  tp → level → tau.
```

level is the security level label to type, tau will be a type with a security label attached to it, tp is a type which has not been assigned with a level label, tau is a labeled tp, rec is a recursive function, one is a unit, nat is a natural number, bool is a boolean, prod is a function which takes 2 types, =>' is a function which takes additional security level for keeping track of implicit leaks and progress, #' is assigning a type with label creating a tau.

References and expressions:

```
exp : type.                          % Expressions
cmd : type.
cont : type.
property : type.
obj : type.
global : obj.
code : property.

true : exp.
false: exp.
unit : exp.
z : exp.
s : exp → exp.
dec : exp → exp.
ref : obj → exp.

fn : (exp → cmd) → exp.
app: exp → exp → exp.
fix : (exp → exp) → exp.
undef : exp.
assign : exp → property → exp → exp.
assign' : property → exp → exp
        = λl. λe. assign (ref global) l e.
not : exp → exp.
== : exp → exp → exp.
new : exp → exp → exp.
```

functionality of expressions will be explained below in their implementation, note that "fn" has an internal function which takes expression as argument and runs a command.

assign' is a sugaring of assign to the global object. code is the code (of commands) which will run inside a constructor.

Commands:

```
# : exp → cmd.
newproperty: (property → cmd) → cmd.
var: exp → property → tau → (obj → exp) → cmd.
var' : property → tau → exp →  cmd
     = λl. λt. λe. var (ref global) l t (λthis. e).
const : exp → property → tau → exp → cmd
     = λo. λl. λt. λe. var o l t (λthis. e).

; : cmd → cmd → cmd.
while: exp → cmd → cmd.
if : exp → cmd → cmd → cmd.
skip: cmd.
return : exp → cmd.
trycatchfinally:
    (cont → cmd) → tau → (exp → cmd) → cmd → cmd.
throw: cont → exp → cmd.
```

var' is a sugaring of var where the scope is the global object, const is a sugaring where this reference is implicit.

Keys and Objects:

```
fields : type.
nil : fields.
field : property → fields → fields.

emptyobj:  exp.

dot : exp → property → exp.
```

nil is the empty field list. dot is predicate that returns the value stored inside a property of an object.

Operational Semantics of expressions:

```
% Deciding equality
equal : exp → exp → exp → type.
equal/u1: equal undef _ false.
equal/u2: equal _ undef false.
equal/z : equal z z true.
equal/s : equal (s E1) (s E2) true
            ← equal E1 E2 true.
neq/z1 : equal z (s E) false.
```

```
neq/z2 : equal (s E) z false.
neq/s  : equal (s E1) (s E2) false
           ← equal E1 E2 false.

neg : exp → exp → type.
neg/true : neg true false.
neg/false: neg false true.
```

"equal" will check if 2 numbers are equal, this is done recursively by calling equal with both number decreased by 1, the final case is returning true in its destination if both numbers reach z simultaneously, if only one number reached z or if one of the expressions is not a number then it will return false in its destination. "neg" (negative) will return the logical opposite of a boolean.

Destinations for destination-based programming:

```
dest : type.
```

Stacks for commands that have to be run:

```
stack : type.
empty : stack.
cons : cmd → stack → stack.
```

where "cons" (constructor) is adding a command to a stack.

Judgments define the operational semantics:

```
eval : exp → dest →  type.
    % Evaluate expression E at destination D
ret : exp → dest → type.
    % Return value V at destination D

run : cmd → stack →  type.
result : exp → type.

store : obj → property → exp → type.

prototype : obj → fields → type.


copy : obj → obj → fields → exp → dest → type.

```

”store” is a predicate for storing values into a property of an object O.F $\mapsto$ V . ”store” is a side effect of the proof derivation, as the language has a state and is not pure.

”prototype” is a predicate for mapping prototypes (field list) in an object, similar to store, prototype is a side effect.

”copy” is a predicate for copying a fieldlist into a new object of the same type (by a constructor).

Reference (ref expression) takes an object and returns an expression. This is done in order to evaluate objects. This is a helping function which is implicit in JS.

```
ref : obj → exp.
```

”dot” is an expression which taking an object and and a field of that object and returning the evaluation of that field (so ”x.y” in JS will be ”dot(x y)” in our syntax).

```
dot : exp → property → exp.
```

Evaluations of expressions:

```
eval/ref : eval (ref O) D —∘
           { ret (ref O) D
         }.

eval/dot : eval (dot E F) D —∘
      { ∃ d.
        eval E d ⊗
        ( Π O.
          ret (ref O) d —∘
          Π X.
        store O F X -@
            { @store O F X ⊗
              ret X D
            }
            )
        }.
```

Evaluating ”ref O” will be consumed and inserted returning in destination D of the object.

Evaluating ”dot E F” will be consumed, a new destination d will be inserted, the expression will be evaluated to d and a new rule will be introduced. Π O . ret (ref O) d —∘ Pi X. store O F X -@ @store O F X ret X D

which means that the expression X which was stored in field F in object O will be be returned in destination D and in parallel expression X will be stored again in property F in object O.

The intuition is that the expression will be evaluated first into the object it contains and then the value X which is stored in this property F will be returned.

```
% Booleans
eval/true : eval true D —∘
            { ret true D
         }.

eval/false: eval false D —∘
      { ret false D
      }.

eval/unit: eval unit D —∘
     { ret unit D
     }.

eval/undef: eval undef D —∘
      { ret undef D
      }.
```

Those are evalutions of axmioms which are consumed into retrning themselves into destiation D.

```
% Natural numbers
eval/z : eval z D —∘
         { ret z D
   }.

eval/s : eval (s E) D —∘
         { ∃ d.
         eval E d ⊗
         ( Π V.
           ret V d —∘
           { ret (s V) D
           }
         )
       }.

eval/dec : eval (dec E) D —∘
        { ∃ d.
          eval E d ⊗
        ( Π V.
          ret (s V) d —∘
          { ret V D
          }
        )
```

```
        }.
```

eval/s is the unnary evaluation of a number which is bigger by 1 from the the argument number E. The evaluation is consumed and replaced by a new parameter d which will be the result of evaluating E and a new linear rule (Π V.ret V d -o ret (s V) D) which is returning the successor of d. The intuition is that exprssion E will be evaluated and then returned the result with 1 added to it.

eval/dec is the unary evaluation of a natural number which is smaller by 1 from the argument. The evaluation is consumed and replaced by a new parameter d which will be the result of evaluating E and a new linear rule (Π V.ret (s V) d -o ret V D) which is returning the V which will be the natural number that was in d less 1 successor and thus that V will returned. The intuition is similar to eval/s.

```

% Assignment
eval/assign : eval (assign E1 F E2) D —∘
        { ∃ d1.
          eval E1 d1 ⊗
            ( Π O.
              ret (ref O) d1 —∘
              { ∃ d2.
                eval E2 d2 ⊗
                ( Π V.
                  ret V d2 —∘
                  Π W.
                  store O F W -@
                  { @store O F V ⊗
                    ret unit D
                  }
                )
              }
            )
        }.
```

assign takes 3 arguments : an expression E1 which is an object, a property F which will be modified in that object, and an expression E2 which will be assigned to that property. the linear rule will first introduce a new parameter d1 which will be the evaluation of the first expression (i.e. object), and a new linear rule (Π O.ret (ref O) d1 —∘ Exists d2. eval E2 d2 ⊗ (Pi V. ret V d2 —∘ Pi W. store O F W -@ @store O F V ⊗ ret unit D)) , the rule will set parameter O to be the object that is returned as d1, that rule will be consumed once into a new parameter d2 which will be the result of evaluating E2 (i.e. the expression to be assigned) and a new linear rule (Π V.ret V d2 —∘ Pi W. store O F W -@ @store O F V ⊗ ret unit D) which will set parameter V to be the result will would have been the expression that would return d2, this will be consumed once

into a new rule Π W. store O F W -@ @store O F V $\otimes$ ret unit D which introduces a new parameter W which will be the result of what was store previously in property F of object O and will be consumed into affine storing parameter V into field F of object O and returning unit into D. The intuition is that E1 will be evaluated into an object, and then the result of evaluating E2 will be stored into the property F of object O.

```
% Testing for equality
eval/== : eval (== E1 E2) D —∘
    { ∃ d1.
      eval E1 d1 ⊗
          ( Π V1.
            ret V1 d1 —∘
            { ∃ d2.
              eval E2 d2 ⊗
            ( Π V2.
              ret V2 d2 —∘
              Π B.
              equal V1 V2 B →
              { ret B D
              }
            )
         }
   )
}.
```

== takes 2 expressions, E1 and E2, and compares them, it does so by introducing new parameter d1, which will be the result of evaluating the first argument into d1, and a new linear rule (Π V1 . ret V1 d1 —∘ Exists d2. eval E2 d2 $\otimes$ (Pi V2. ret V2 d2 —∘ Pi B. equal V1 V2 B → ret B D) which introduces a new parameter V1 which will be the expression that would have returned d1. this is consumed into a new parameter d2 which is the evaluation of the second argument and a new linear rule (Π V2 . ret V2 d2 —∘ Pi B. equal V1 V2 B → ret B D) which introduces a new parameter V2 which is what d2 will return, this will be consumed into a new rule which introduces a new parameter B which will be the result of equal V1 V2 B (see above) and this will imply that B will be returned in destination D. The intuition is that the that "equal" will be run with the results of evaluating E1 and E2 as arguments, and the result will be returned in the destination.

```
% Computing the Boolean complement
eval/not : eval (not E) D —∘
       { ∃ d.
       eval E d ⊗
       ( Π V.
         Π V'.
         ret V d —∘
         neg V V' →
```

```
          { ret V' D
          }
        )
      }.
```

not is taking 1 argument which is a boolean expression and returns the logical opposite of that argument. It does so by introducing a new parameter d which will be the result of evaluating the argument and a new rule (Π V.Pi V'. ret V d —∘ neg V V' → ret V' D) which is introducing 2 parameters V and V'. V will be the expression which d will be returned from. this will be consumed into evaluating V' as the result of neg (negative - see above) and this implies that V' will be returned into D. The intuition is that "neg" will run with the evaluation of expression E as an argument and its result will be returned in the destination.

```
eval/empty: eval emptyobj D —∘
       { ∃ o:obj.
         @prototype o nil ⊗
         ret (ref o) D
       }.

eval/new : eval (new E1 E2) D —∘
     { ∃ d.
       eval E1 d ⊗
       ( Π O.
         ret (ref O) d —∘
         Π Fs.
         prototype O Fs -@
         { @prototype O Fs ⊗
           ∃ o:obj.
           @prototype o Fs ⊗
           copy O o Fs E2 D
         }
       )
     }.
```

"emptyobj" is the axiom of en empty object, it does so by introducing a new parameter o of kind obj, affine prototyping nil into o and returning o in D. The intuition is that there will be stored some object with empty field list and that object will be returned at the destination.

"new" is creating and a returning a new object of the first argument E1 with the argument E2 , it does so by introducing a new parameter d which is the result of the evaluation of E1 and a new rule (Π O.ret (ref O) d —∘ Pi Fs. prototype O Fs -@ @prototype O Fs ⊗ Exists o:obj. @prototype o Fs ⊗ copy O o Fs E2 D)

which introduces a new parameter O which is the object that would result into d, this will be replaces by introducing a new parameter Fs which is the set of labels which will was assigned to O through prototype, this will be consumed into the mupltiple applied set of facts { @prototype O Fs ⊗ Exists o:obj. @prototype o Fs ⊗ copy O o Fs E2 D the first fact is assigning Fs to O again through prototype, the second one introduces a new parameter o which will be the result of a prototype with Fs and making a copy (see below) of O, o, Fs, E2 into D. The intuition is that expression E1 will be evaluated into its object, and and "copy" (see next rule) will be called with that object, its field list, a new object and expression E2 (as the argument to the constructor).

```
copy/nil : copy O O' nil E2 D —∘
     { eval (app (app (dot (ref O') code) (ref O')) E2)  D}.

copy/field : copy O O' (field F Fs) E2 D —∘
        Π V.
        store O F V -@
        { @store O F V ⊗
          @store O' F V ⊗
          copy O O' Fs E2 D
        }.
```

The way that copy works is by traversing field by field, the base case is when the list of fields is empty (nil). Thus the first case copy/nil is copying O to O' by initiating the class O with the argument E2. When that field list is nil it means that all fields has already been assigned to O'. And thus O' is now containing all the fields is meant to and now all left to to run the constructor which is stored in "code". And thus "code" is run first with the newly created object O' in order to assign its next actions with it, and the result of that is run with the argument E2 which is causing the newly created object O' to contain all what the constructor O is assigning to it with argument E2. The intuition is that the constructor is always made of 2 nested functions, as there could only be one new variable per function. Here, we need 2 new variables. As one is the variable of the argument to the constructor and one is the newly created object. The first function will evaluate E2 and return the second function with evaluated E2 as a new variable, then the second will be run (as it has evaluated E2 as a variable, because it's a nested function) with a newly created object as a variable (the idea is that the final function of the constructor should have both the argument and the new object as variable, so it could assign new properties with the argument to the new object), and the result will be evaluated into destination D (see example in the end of this section).

The case when the field list Fs is not empty, first Fs will be split to the first field F and the rest of the fields Fs, then copy will be recursively called with the rest of Fs after adding one field F to the new object O' . This is done in linear logic by creating a new parameter V which will be the what was stored in field F in object O, this then will be consumed into 3 parallel facts, affine storing V to field F of object O, affine storing O' to field F of the new object O' and recursively calling copy again with the rest of the field list Fs.

```
% Functions
eval/fn : eval (fn C) D ⊸
    { ret (fn C) D
    }.


% Application
eval/app : eval (app E1 E2) D ⊸
     { ∃ d1.
       eval E1 d1 ⊗
       ∃ d2.
       eval E2 d2 ⊗
       ( Π C.
         ret (fn C) d1 ⊸
         Π V.
         ret V d2 ⊸
         { run (C !V) empty ⊗
         ( Π W.
           result W ⊸
           { ret W D
           }
     )
    }
  )
 }.

% Recursive named functions
eval/fix : eval (fix E) D ⊸
     { eval (E !(fix E)) D}.
```

eval/fn takes as argument the command that should be run once the function is called, declaring a function doesn't actually evaluates the function and thus will only return the function itself into destination D.

eval/app takes a function E1 and an argument for that function E2 and evaluated that function with that function with input E2 into destination D. This is done in linear logic through introducing a new parameter d1 which will be the result of evaluating E1, and a new parameter d2 which will be the result evaluating E2 and a new rule (Π C.ret (fn C) d1 ⊸ Pi V. ret V d2 ⊸ run (C !V) empty ⊗ (Pi W. result W ⊸ ret W D))

This rule will introduce a new parameter C which will be the command which is contained in the function which will return d1, this is then consumed into the new rule which introduces a new parameter V which will be that expression which will return

d2, this will be consumed into a new fact which will run C with input V in an empty stack and introduce a new parameter W which will be what is returned from running C this will be consumed into returning W into destination D. The intuition is that E1 will be evaluated into a function and then the command which is the content of that function will run with the result of evaluating E2 as input.

eval/fix is the declaration of a recursive function which will evaluate its argument E with the input of itself !(fix E), since it is recursive, it could be consumed many times and hence the !.

```
run/newproperty: run (newproperty C) CS ⊸
    { ∃ f:property.
      run (C !f) CS
    }.

% Variable declaration
run/var : run (var E1 L T E2) CS ⊸
          { ∃ d1.
          eval E1 d1 ⊗
          ( Π O.
            ret (ref O) d1 ⊸
            Π Fs.
            prototype O Fs -@
            { ∃ d2.
              eval (E2 !O) d2 ⊗
            ( Π V2.
              ret V2 d2 ⊸
              { @store O L V2 ⊗
                @prototype O (field L Fs) ⊗
                run skip CS
              }
            )
        }
  )
}.
```

run/newproperty is taking a command and is running that command with a new property name in it.

run/var is taking a an object as expression E1, a property L, a leveled type T and an expression to be assigned E2. Then it will assign E2 (which is expected at labeled type T) to property L of object E1. This will be done by introducing a new parameter d1 which will be the result of evaluating E1, and introducing a new rule (Π O.ret (ref O) d1 ⊸ Pi Fs. prototype O Fs -@ Exists d2. eval (E2 !O) d2 ⊗ (Pi V2. ret V2 d2 -o @store O L V2 ⊗ @prototype O (field L Fs) ⊗ run skip CS))

which is introducing a new variable O that will be the object that would return d1, this will be then consumed into a new a new rule which introduces a new parameter Fs

which will be the list of properties that O would have been related to. this then will be consumed into a rule which will introduce a new parameter d2. which will be the result of evaluating E2 with the object O and a new rule which introduces a new parameter V2 which will be the expression which will return d2, this then will be consumed into storing V2 in property L of object O, and relating the field list Fs together with L to object O and running skip in the command stack. The intuition is that E1 will be evaluated into an object, then E2 will be evaluated, and the result of E2 will be stored in the property L of the object of E1.

```
% Sequencing of commands
run/;  : run (; C1 C2) CS ⊸
    { run C1 (cons C2 CS)
   }.

% Evaluate an epression embedded in a command
run/#  : run (# E) (cons C CS) ⊸
        { ∃ d.
        eval E d ⊗
         ( Π V. ret V d ⊸
          { run C CS
         }
       )
   }.
```

run/; takes 2 commands, C1 and C2, as arguments, runs the first one C1, and then the second one C2. It does so by running the first one with the new command stack which includes C2 as the first command.

run/ takes an expression E as an argument, and evaluates it on the run as if it's a command. It is done by introducing a new parameter d, which will be the evaluation result of the expression E, and a new rule `( Π V. ret V d ⊸` run C CS `)` which is introducing a new parameter V which will be the expression that would return d, and consumes it into running the next command in the command stack.

```
% Evaluate skip (either pick the next element from the stack or end
run/skip1 : run skip (cons C CS) ⊸
      { run C CS
      }.

run/skip2 : run skip empty ⊸
      { result unit
      }.

run/return : run (return E) CS ⊸
      { ∃ d.
        eval E d ⊗
        ( Π V.
```

```
          ret V d ⊸
    { result V
    }
   )
 }.
```

run/skip is the command which doesn't do anything. This command has 2 cases, one in which the command stack is empty and the second in which the command stack isn't empty. In the the command stack is empty, the program will terminate resulting a unit, in case the command stack isn't empty, the next command in the stack will run.

run/return will take one expression E as an argument and will return it as a result of the clause it's run in. It will do so by introducing a new parameter d which will be the result of evaluating E and a new rule (Π V. ret V d ⊸ result V) which is introducing a new parameter V which will be the the value that would return d, and then it will be consumed into result V.

```

% Conditionals and While loops
exec : exp → cmd → cmd → stack → type.  % Auxiliary judgments
exec' : exp → exp → cmd → stack → type.

run/if : run (if E C1 C2) CS ⊸
         { ∃ d.
     eval E d ⊗
     ( Π V.
       ret V d ⊸
       { exec V C1 C2 CS
       }
     )
   }.

run/while : run (while E C1) CS ⊸
      { ∃ d.
        eval E d ⊗
        ( Π V.
          ret V d ⊸
    { exec' V E C1 CS
    }
  )
}.

run/exec/true : exec true C1 C2 CS ⊸
          { run C1 CS
    }.

```

```
run/exec/false : exec false C1 C2 CS —∘
           { run C2 CS
     }.

run/exec'/true : exec' true E C1 CS —∘
           { run C1 (cons (while E C1) CS)
     }.

run/exec'/false : exec' false E C1 CS —∘
      { run skip CS
      }.

```

run/if will take 3 arguments, a Boolean expression E, and 2 commands C1 and C2, then it will evaluate E, in case E evaluates to true, C1 will run, and in case E is false C2 will run. It will do so by introducing a new parameter d which will be the result of evaluating E and a new rule (Π V.ret V d —∘ exec V E C1 C2 CS) which is introducing a new parameter v which will be that expression that will return d, this is then consumed into running exec with arguments V, E, C1, C2 and CS.

run/while will take 3 arguments, a Boolean expression E, and a command C, then it will evaluate E, in case E evaluates to true, C will run followed by the same call on while, It will do so by introducing a new parameter d which will be the result of evaluating E and a new rule (Π V.ret V d —∘ exec' V E C CS) which is introducing a new parameter v which will be that expression that will return d, this is then consumed into running exec' with arguments V, E, C and CS.

exec takes a boolean expression, a command C1 and a command C2, in case the expression is true run/exec/true will run and this command C1 is run with CS command stack. in case the expression is false run/exec/false will run and this command C1 is run with CS command stack.

exec' takes a boolean expression, an expression E and command C1, in case the boolean expression is true run/exec'/true will run and thus command C1 is run with the command (while E C1) appended to CS command stack. in case the boolean expression is false, run/exec'/false will run and thus command skip is run with CS command stack.

```
% Exception Handling
sleep : cont → (exp → cmd) → stack → type.
    % auxiliary judgment
```

"Try catch finally" is a command which takes 3 command, it will run the first command C_1, in case it throws an exception, it will run the catch clause C_2 with the expression which was thrown as an exception, and finally it will run C_3.

For the implementation we would use a continuation parameter $k : cont$ which will point out where the the throw is, and will clarify which catch clause is this throw related

to.

In line 2 the judgment for sleep is presented, the sleep is the optional continuation of the the run of C_1 in case an exception has been thrown. Sleep takes 3 arguments: the exception k which has been thrown, the catch clause C_2 (which also includes the expression that has been thrown) and the rest of the command stack.

Recall the commands trycatchfinally and throw:

```
1 trycatchfinally:
2     (cont → cmd) → tau → (exp → cmd) → cmd → cmd.
3 throw: cont → exp → cmd.
```

Where trycatchfinally takes 3 commands arguments as described above and a tau (labeled type) which will be the expected type of the exception. Notice that the first command (which is the try clause) includes the continuation that could be thrown in it, and the second command (the catch clause) includes the expression that has been thrown.

Recall throw (line 3) which takes the continuation point and the expression that is thrown.

The following list shows the linear logic rules for the implementation of those commands:

```
1  run/trycatchfinally : run (trycatchfinally C1 _ C2 C3) CS —∘
2          { ∃ k.
3            run (C1 !k) (cons C3 CS) ⊗
4          @sleep k C2 (cons C3 CS)
5          }.
6
7  run/throw : run (throw K E) _ —∘
8        sleep K C2 CS -@
9        { ∃ d.
10         eval E d ⊗
11         ( Π V.
12           ret V d —∘
13             run (C2 !V) CS
14         )
15       }.
```

The fact `run (trycatchfinally C1 _ C2 C3) CS`

will be then consumed and instead a new parameter k and the following 2 facts will be inserted: `run (C1 !k) (cons C3 CS)` and `@sleep k C2 (cons C3 CS)` . Note that sleep is an affine assumption market by @ and doesn't have to be consumed. The next command `run/throw :` of the throw command. It will be consumed and then a `sleep K C2 CS` fact will also be consumed, and replaced by a new destination d, a new fact `eval E d` and a new rule Π `V.`ret V d —∘ run (C2 !V) CS . The new rule

is linear and will run once since rule `eval V` will only return one V. The intuition is that the sleep called will be in the stack of the memory as long as C1 runs, and once throw is called that sleep will be invoked with expression E of the throw evaluated as its input (and C3 from the trycatchfinally will run afterwards in any case).

Examples: Example for constructor using ”new” and ”function”:
the following JS program:

```
1 function Person(x) {
2    this.name = x;
3 }
4 var alice = new Person(1);
5 var bob = new Person(5);
6 return(alice.name);
```

Corresponds to the following Celf program:

```
1  constructor : cmd
2   = newproperty λperson.
3     newproperty λname.
4     newproperty λalice.
5     newproperty λbob.
6     (; (function !(ref global) !person  !(#' rec low)
7          !(λthis. fn λobj. return (fn λx.
8                   ; (var obj name !(#' nat low) (λthis. x))
9                   (return obj))))
10    (; (var' !alice !(#' nat low) !(new (dot' !person) (s z)))
11    (; (var' !bob !(#' nat low) !(new (dot' !person)
12                                    (s (s (s (s (s z)))))))
13    (return (dot (dot' !alice) name))))).
```

The constructor will always be of the form : \!this. fn \!obj. return (fn \!x. code) where ”this” refers to the function object of the constructor, ”obj” refers to the newly created object by new and ”x” refers to the argument passed to the constructor. in this example the construtor (denoted as ”function”) will create a new ”person” with its property ”name” assignemed with the argument ”x”. This is done by declaring ”person” to be a function which returns a function that takes argument ”obj” and returns a function that takes argument ”x” and initiates a new property in the object ”obj” called ”name” of labeled type (nat low). As can be seen in lines 6-8, the constructor is made of 2 functions, the outed which introduces the new variable ”obj” and the inner which introduces ”x”, where obj is the new object to be created and x is the input argument. Thus calling later the ”new” command (line 10) will run both functions and return a new object with its properties assigned according to the different input.

Running that program the following query:

```
#query 1000 ⊗ ⊗ 1 (prototype global nil -@ run constructor
                                          empty —∘ {result V}).
```

Which means that the command ”constructor” will run under the empty global object with the initial empty command stack and will be consumed into an result ”V”.

Celf will return this result after running this query:

```
Solution: \@X1. \X2. {
    let {[!f, X3]} = run/newproperty X2 in
    let {[!f_1, X4]} = run/newproperty X3 in
    let {[!f_2, X5]} = run/newproperty X4 in
    let {[!f_3, X6]} = run/newproperty X5 in
    let {X7} = run/; X6 in
    let {X8} = run/; X7 in
    let {[!d1, [X9, X10]]} = run/var X8 in
    let {X11} = eval/ref X9 in
    let {[!d2, [X12, X13]]} = X10 !global X11 !nil @X1 in
    let {[!o, [@X14, X15]]} = eval/empty X12 in
    let {[@X16, [@X17, X18]]} = X13 !(ref !o) X15 in
    let {X19} = run/skip1 X18 in
    let {[!d1_1, [X20, X21]]} = run/var X19 in
    let {[!d, [X22, X23]]} = eval/dot X20 in
    let {X24} = eval/ref X22 in
    let {[@X25, X26]} = X23 !global X24 !(ref !o) @X16 in
    let {[!d2_1, [X27, X28]]} = X21 !o X26 !nil @X14 in
    let {X29} = eval/fn X27 in
    let {[@X30, [@X31, X32]]} = X28 !(fn !(\!X30. return !(fn !(\!x. ; !(var
    !X30 !f_1 !(#’ !nat !low) !(\!this. x)) !(return !X30))))) X29 in
    let {X33} = run/skip1 X32 in
    let {X34} = run/; X33 in
    let {[!d1_2, [X35, X36]]} = run/var X34 in
    let {X37} = eval/ref X35 in
    let {[!d2_2, [X38, X39]]} = X36 !global X37 !(field !f !nil) @X17 in
    let {[!d_1, [X40, X41]]} = eval/new X38 in
    let {[!d_2, [X42, X43]]} = eval/dot X40 in
    let {X44} = eval/ref X42 in
    let {[@X45, X46]} = X43 !global X44 !(ref !o) @X25 in
    let {[@X47, [!o_1, [@X48, X49]]]} = X41 !o X46 !(field !code !nil) @X31 in
    let {[@X50, [@X51, X52]]} = copy/field X49 !(fn !(\!X50. return !(fn
    !(\!x. ; !(var !X50 !f_1 !(#’ !nat !low) !(\!this. x)) !(return !X50))))) @X30 in
    let {X53} = copy/nil X52 in
    let {[!d1_3, [X54, [!d2_3, [X55, X56]]]]} = eval/app X53 in
    let {[!d_3, [X57, X58]]} = eval/s X55 in
    let {[!d1_4, [X59, [!d2_4, [X60, X61]]]]} = eval/app X54 in
```

```
let {X62} = eval/z X57 in
let {X63} = eval/ref X60 in
let {[!d_4, [X64, X65]]} = eval/dot X59 in
let {X66} = eval/ref X64 in
let {[@X67, X68]} = X65 !o_1 X66 !(fn !(\!X67. return !(fn !(\!x. ; !(var
!X67 !f_1 !(#' !nat !low) !(\!this. x)) !(return !X67))))) @X51 in
let {X69} = X58 !z X62 in
let {[X70, X71]} = X61 !(\!X70. return !(fn !(\!x. ; !(var !X70 !f_1
!(#' !nat !low) !(\!this. x)) !(return !X70)))) X68 !(ref !o_1) X63 in
let {[!d_5, [X72, X73]]} = run/return X70 in
let {X74} = eval/fn X72 in
let {X75} = X73 !(fn !(\!X75. ; !(var !(ref !o_1) !f_1 !(#' !nat !low)
!(\!this. X75)) !(return !(ref !o_1)))) X74 in
let {X76} = X71 !(fn !(\!X76. ; !(var !(ref !o_1) !f_1 !(#' !nat !low)
!(\!this. X76)) !(return !(ref !o_1)))) X75 in
let {[X77, X78]} = X56 !(\!X77. ; !(var !(ref !o_1) !f_1 !(#' !nat !low)
!(\!this. X77)) !(return !(ref !o_1))) X76 !(s !z) X69 in
let {X79} = run/; X77 in
let {[!d1_5, [X80, X81]]} = run/var X79 in
let {X82} = eval/ref X80 in
let {[!d2_5, [X83, X84]]} = X81 !o_1 X82 !(field !code !nil) @X48 in
let {[!d_6, [X85, X86]]} = eval/s X83 in
let {X87} = eval/z X85 in
let {X88} = X86 !z X87 in
let {[@X89, [@X90, X91]]} = X84 !(s !z) X88 in
let {X92} = run/skip1 X91 in
let {[!d_7, [X93, X94]]} = run/return X92 in
let {X95} = eval/ref X93 in
let {X96} = X94 !(ref !o_1) X95 in
let {X97} = X78 !(ref !o_1) X96 in
let {[@X98, [@X99, X100]]} = X39 !(ref !o_1) X97 in
let {X101} = run/skip1 X100 in
let {X102} = run/; X101 in
let {[!d1_6, [X103, X104]]} = run/var X102 in
let {X105} = eval/ref X103 in
let {[!d2_6, [X106, X107]]} = X104 !global X105 !(field !f_2 !(field !f
!nil)) @X99 in
let {[!d_8, [X108, X109]]} = eval/new X106 in
let {[!d_9, [X110, X111]]} = eval/dot X108 in
let {X112} = eval/ref X110 in
let {[@X113, X114]} = X111 !global X112 !(ref !o) @X45 in
let {[@X115, [!o_2, [@X116, X117]]]} = X109 !o X114 !(field !code !nil) @X47
in
let {[@X118, [@X119, X120]]} = copy/field X117 !(fn !(\!X118. return !(fn
!(\!x. ; !(var !X118 !f_1 !(#' !nat !low) !(\!this. x)) !(return !X118))))) @X50 in
let {X121} = copy/nil X120 in
```

```
let {[!d1_7, [X122, [!d2_7, [X123, X124]]]]} = eval/app X121 in
let {[!d1_8, [X125, [!d2_8, [X126, X127]]]]} = eval/app X122 in
let {[!d_10, [X128, X129]]} = eval/dot X125 in
let {X130} = eval/ref X128 in
let {[@X131, X132]} = X129 !o_2 X130 !(fn !(\!X131. return !(fn !(\!x. ;
!(var !X131 !f_1 !(#' !nat !low) !(\!this. x)) !(return !X131))))) @X119 in
let {X133} = eval/ref X126 in
let {[X134, X135]} = X127 !(\!X134. return !(fn !(\!x. ; !(var !X134 !f_1
!(#' !nat !low) !(\!this. x)) !(return !X134)))) X132 !(ref !o_2) X133 in
let {[!d_11, [X136, X137]]} = eval/s X123 in
let {[!d_12, [X138, X139]]} = run/return X134 in
let {X140} = eval/fn X138 in
let {X141} = X139 !(fn !(\!X141. ; !(var !(ref !o_2) !f_1 !(#' !nat !low)
!(\!this. X141)) !(return !(ref !o_2)))) X140 in
let {[!d_13, [X142, X143]]} = eval/s X136 in
let {X144} = X135 !(fn !(\!X144. ; !(var !(ref !o_2) !f_1 !(#' !nat !low)
!(\!this. X144)) !(return !(ref !o_2)))) X141 in
let {[!d_14, [X145, X146]]} = eval/s X142 in
let {[!d_15, [X147, X148]]} = eval/s X145 in
let {[!d_16, [X149, X150]]} = eval/s X147 in
let {X151} = eval/z X149 in
let {X152} = X150 !z X151 in
let {X153} = X148 !(s !z) X152 in
let {X154} = X146 !(s !(s !z)) X153 in
let {X155} = X143 !(s !(s !(s !z))) X154 in
let {X156} = X137 !(s !(s !(s !(s !z)))) X155 in
let {[X157, X158]} = X124 !(\!X157. ; !(var !(ref !o_2) !f_1 !(#' !nat !low)
!(\!this. X157)) !(return !(ref !o_2))) X144 !(s !(s !(s !(s !(s !z))))) X156 in
let {X159} = run/; X157 in
let {[!d1_9, [X160, X161]]} = run/var X159 in
let {X162} = eval/ref X160 in
let {[!d2_9, [X163, X164]]} = X161 !o_2 X162 !(field !code !nil) @X116 in
let {[!d_17, [X165, X166]]} = eval/s X163 in
let {[!d_18, [X167, X168]]} = eval/s X165 in
let {[!d_19, [X169, X170]]} = eval/s X167 in
let {[!d_20, [X171, X172]]} = eval/s X169 in
let {[!d_21, [X173, X174]]} = eval/s X171 in
let {X175} = eval/z X173 in
let {X176} = X174 !z X175 in
let {X177} = X172 !(s !z) X176 in
let {X178} = X170 !(s !(s !z)) X177 in
let {X179} = X168 !(s !(s !(s !z))) X178 in
let {X180} = X166 !(s !(s !(s !(s !z)))) X179 in
let {[@X181, [@X182, X183]]} = X164 !(s !(s !(s !(s !(s !z))))) X180 in
let {X184} = run/skip1 X183 in
let {[!d_22, [X185, X186]]} = run/return X184 in
```

```
    let {X187} = eval/ref X185 in
    let {X188} = X186 !(ref !o_2) X187 in
    let {X189} = X158 !(ref !o_2) X188 in
    let {[@X190, [@X191, X192]]} = X107 !(ref !o_2) X189 in
    let {X193} = run/skip1 X192 in
    let {[!d_23, [X194, X195]]} = run/return X193 in
    let {[!d_24, [X196, X197]]} = eval/dot X194 in
    let {[!d_25, [X198, X199]]} = eval/dot X196 in
    let {X200} = eval/ref X198 in
    let {[@X201, X202]} = X199 !global X200 !(ref !o_1) @X98 in
    let {[@X203, X204]} = X197 !o_1 X202 !(s !z) @X89 in
    let {X205} = X195 !(s !z) X204 in X205}
 #V = s !z
```

So the result V will be 1 (i.e. ”s z”) which is correct.

Example: Doubling. A recursive function (as an object):

```
function double(x) {
  if (x==0) {return(0);}
  else {return (2+double(x-1));}
}
return (double(3));
```

corresponds to the following Celf program:

```
double : cmd
 = newproperty λdouble.
   (; (function !(ref global) !double !(#' rec low)
            !(λthis. fix λdouble. fn λx.
         if (== x z)  (return z)
                        (return (s (s (app double (dec x)))))))
   (return (fapp !(ref global) !double !(s (s (s z)))))).
```

Example: Try catch finally

```
var x;
var y;
try {
  throw (function (x) {return (x+1);}); x = 0; y=3;
}
catch (f) {
  x = f(1);
}
finally {
  y = x;
```

```
}
return(y);
```

```
advtrycatch
  : cmd
  = newproperty λx.
    newproperty λy.
    (; (var' !x !(#' nat low) !undef)
    (; (var' !y !(#' nat low) !undef)
    (; (trycatchfinally (λk. (; (throw k (fn λx. return (s !x)))
           (; (# (assign' !x !z))
           (; (# (assign' !y !(s (s (s z)))))
           skip))))
        (#' nat low) (λf. (# (assign' !x !(app f (s z)))))
        (# (assign' !y !(dot' !x))))
    (return (dot' !y))))).
```

7 Type System

In the type system (see below) we will differentiated between types (”T”) and labeled types (denoted ”θ”), since in our system, it is important that every type is labeled during the evaluation process. The deconstruction of the labeled types is necessary for the comparison of the labels. There are no types in the JavaScript core syntax, however we introduce them here in JSL for the ease of detecting error prior to run time and for it is easier to construct type derivations this way.

7.1 Type System Rules

Recall that in the scope of this paper, we only worked with a simple security level hierarchy of 2 levels only. Namely high (”H”) and low (”L”). In a future work this hierarchy could be expended to a wider lattice, though the type logic will be the same. Types are denoted as T. 1 is the type of a unit. Nat is the type of a natural number. Bool is the type of a boolean. $\theta is a type labeled with a security level.\mathrm{l}_1;\theta_1 \rightarrow^{l_2} \theta_2$ is the type of a function, the first level is the blocking level, the second level is the side effects track, the first θ is the type of the input and the second θ is the type of the output.

Similarly to the operational semantics, the type system is a set of logical rules. These rules will assure that all running programs are safe by logically deriving to axioms. ”Γ” is an identifier typing. An identifier typing is a map from identifiers to labeled types; it gives the types of any free identifiers of e, checking whether e is well typed to a certain given type (security level ”L” (low) of type ”T” in this case). $\vdash$ means well typed, ”pc” is the program counter tracking implicit flows and ”γ” is the

security level lattice. “,” is the operator to extend Gamma with. Note that command and expressions have an extra level (”bl” for blocking level) which assures PSNI.

Rules for expressions:

$$\frac{}{bl;\Gamma,x:\theta\vdash_{pc} x:\theta}\ \text{axiom}$$

$$\frac{}{bl;\Gamma\vdash_{pc} true:bool^{l}}\ \text{of/true}$$

$$\frac{}{bl;\Gamma\vdash_{pc} false:bool^{l}}\ \text{of/false}$$

$$\frac{}{bl;\Gamma\vdash_{pc} ():one^{l}}\ \text{of/unit}$$

$$\frac{}{bl;\Gamma\vdash_{pc} z:nat^{l}}\ \text{of/z}$$

$$\frac{bl;\Gamma\vdash_{pc} e:nat^{l}}{bl;\Gamma\vdash_{pc} (s\ e):nat^{l}}\ \text{of/s}$$

$$\frac{bl;\Gamma\vdash_{pc} e:nat^{l}}{bl;\Gamma\vdash_{pc} e:nat^{l}}\ \text{of/dec}$$

$$\frac{bl;\Gamma,x:\theta_1\vdash_{le} c:\theta_2}{bl;\Gamma\vdash_{pc} \lambda x.c:(bl;\theta_1\rightarrow^{le}\theta_2)^{Low}}\ \text{of/fn}$$

$$\frac{bl;\Gamma,f:\theta_1\vdash_{le} c:\theta_2}{bl;\Gamma\vdash_{pc} \mu f.c:(bl\sqcup le;\theta_1\rightarrow^{le}\theta_2)^{Low}}\ \text{of/fix}$$

$$\frac{bl;\Gamma\vdash_{pc} e_1:(bl;\theta_1\rightarrow^{le} T_2^{l_2})^{l_3}\quad bl;\Gamma\vdash_{pc} e_2:\theta_1\quad \gamma\vdash l_3\sqsubseteq l_2\quad \gamma\vdash pc\sqcup l_3\sqsubseteq le}{bl;\Gamma\vdash_{pc} e_1e_2:T_2^{l_2}}\ \text{of/app}$$

$$\frac{}{bl;\Gamma\vdash_{pc} \bot:nat^{l}}\ \text{of/undef}$$

$$\frac{\Gamma\vdash v:T_1^{l_2}\quad \gamma\vdash pc\sqsubseteq l_2\quad \gamma\vdash l_2\sqsubseteq l_1\quad \gamma\vdash l_3\sqsubseteq l_2\quad bl_1;\Gamma\vdash_{pc} e_1:objectable_1^{l_1}\quad bl_2;\Gamma\vdash_{pc} e_2:T_2^{l_3}\quad \gamma\vdash bl_1\sqsubseteq bl_2}{bl_2;\Gamma\vdash_{pc} e_1.v:=e_2:T_1^{l_2}}\ \text{of/assign}$$

$$\frac{bl;\Gamma\vdash_{pc} e:bool^{l}}{bl;\Gamma\vdash_{pc} \neg e:bool^{l}}\ \text{of/not}$$

$$\frac{\Gamma\vdash o:T^{l}}{bl;\Gamma\vdash_{pc} !o:T^{l}}\ \text{of/ref}$$

$$\frac{bl_1;\Gamma\vdash_{pc} e_1:T_1^{l_1}\quad bl_2;\Gamma\vdash_{pc} e_2:T_2^{l_2}\quad \gamma\vdash bl_1\sqsubseteq bl_2}{bl_2;\Gamma\vdash_{pc} e_1=e_2:bool^{l_1\sqcup l_2}}\ \text{of/==}$$

Rules for commands:

$$\frac{bl;\Gamma \vdash_{pc} e : \theta}{bl;\Gamma \vdash_{pc} c : \theta}\ \text{of/\#}$$

$$\frac{bl_1;\Gamma,o : T_1^{l_1} \vdash_{pc} e_2 : T_2^{l_2} \quad bl_2;\Gamma \vdash_{pc} e_1 : T_3^{l_3} \quad \Gamma \vdash v : T_1^{l_1} \quad \gamma \vdash l_3 \sqsubseteq l_1 \quad \gamma \vdash l_2 \sqsubseteq l_1 \quad \gamma \vdash pc \sqsubseteq l_1 \quad \gamma \vdash bl_1 \sqsubseteq bl_2}{bl_2;\Gamma \vdash_{pc} \text{var } e_1.v : T_2^{l_2} = o \mapsto e_2}\ \text{of/var}$$

$$\frac{bl_1;\Gamma \vdash_{pc} c_1 : T_1^{l_2} \quad bl_2;\Gamma \vdash_{pc} c_2 : T_2^{l_3} \quad \gamma \vdash bl_1 \sqsubseteq bl_2}{bl_2;\Gamma \vdash_{pc} \ c_1 \ ; \ c_2 : T_2^{l_2 \sqcup l_3}}\ \text{of/;}$$

$$\frac{l_1;\Gamma \vdash_{pc} e : bool^{l_2} \quad l_1;\Gamma \vdash_{pc \sqcup l_2} c : T^{l_3} \quad \gamma \vdash l_2 \sqsubseteq l_1 \quad \gamma \vdash l_2 \sqsubseteq l_3}{l_1;\Gamma \vdash_{pc} while(e)\ \{c\} : T^{l_3}}\ \text{of/while}$$

$$\frac{bl_1;\Gamma \vdash_{pc} e : bool^{l_1} \quad bl_2;\Gamma \vdash_{pc \sqcup l_1} c_1 : T^{l_2} \quad bl_2;\Gamma \vdash_{pc \sqcup l_1} c_2 : T^{l_2} \quad \gamma \vdash l_2 \sqsubseteq l_1 \quad \gamma \vdash bl_1 \sqsubseteq bl_2}{bl_2;\Gamma \vdash_{pc} if(e)\ \{c_1\}\ else\ \{c_2\} : T^{l_2}}\ \text{of/if}$$

$$\frac{}{bl;\Gamma \vdash_{pc} skip : \theta}\ \text{of/skip}$$

$$\frac{\gamma \vdash pc_1 \sqsubseteq pc_2 \quad bl_1;\Gamma,k_{\theta_1} : \theta_2 \vdash_{pc_1} k.c_1 : \theta_2 \quad bl_2;\Gamma,e : \theta_1 \vdash_{pc_2} e.c_2 : \theta_2 \quad bl_3;\Gamma_{pc_2} c_3 : \theta_2 \quad \gamma \vdash bl_1 \sqsubseteq bl_2 \sqsubseteq bl_3}{bl_3;\Gamma \vdash_{pc_1} \text{try}\{c_1,\theta_1\} \text{ catch } \{c_2\} \text{ finally } \{c_3\} : \theta_2}\ \text{of/trycatchfinally}$$

$$\frac{\Gamma \vdash_{pc} k_{\theta_1} : \theta_2 \quad bl;\Gamma \vdash_{pc} e : \theta_1}{bl;\Gamma \vdash_{pc} throw(k,\ e) : \theta_2}\ \text{of/throw}$$

$$\frac{bl;\Gamma \vdash_{pc} e : T^l}{bl;\Gamma \vdash_{pc} return(e) : T^l}\ \text{of/return}$$

Rules for the object system:

$$\frac{bl_1;\Gamma \vdash_{pc} e_1 : objectable^{l_1} \quad bl_2;\Gamma \vdash_{pc} e_2 : T^{l_2} \quad \gamma \vdash l_2 \sqsubseteq l_1 \quad \gamma \vdash bl_1 \sqsubseteq bl_2}{bl_2;\Gamma \vdash_{pc} new\ e_1(e_2) : T_1^l} \text{ of/new}$$

$$\frac{\Gamma \vdash v : T^{l_2} \quad \Gamma \vdash fs : fieldlist^{l_1}}{\Gamma \vdash field(v,\ fs) : fieldlist^{l_1 \sqcup l_2}} \text{ of/field}$$

$$\frac{}{\Gamma \vdash nil : T^{Low}} \text{ of/nil}$$

$$\frac{\Gamma \vdash v : T^l \quad bl;\Gamma \vdash_{pc} e : objectable^l}{bl;\Gamma \vdash_{pc} e.v : T^l} \text{ of/dot}$$

$$\frac{bl;\Gamma, v : T^{l_2} \vdash_{pc} c : T^{l_1} \quad \gamma \vdash l_2 \sqsubseteq l_1}{bl;\Gamma \vdash_{pc} newproperty(v.c) : T^{l_1}} \text{ of/newproperty}$$

Rules for outputs (not written in JSL yet):

$$\frac{Low;\Gamma \vdash_{Low} e : T^{Low}}{Low;\Gamma \vdash_{Low} outL(e) : unit^{Low}} \text{ of/outL}$$

$$\frac{}{bl;\Gamma \vdash_{pc} outH(e) : unit^{Low}} \text{ of/outH}$$

Rules for subtyping:

$$\frac{}{subtp(T\ T)} \text{ subtp/type}$$

$$\frac{l_1 \sqsubseteq l_2 \quad subtp(T_1\ T_2)}{subtau(T^{l_1}\ T^{l_2})} \text{ subtau/\#}$$

$$\frac{l_2 \sqsubseteq l_1}{sublevel(l_1\ l_2)} \text{ sublevel}$$

$$\frac{subtau(\theta_1'\ \theta_1) \quad subtau(\theta_2\ \theta_2') \quad sublevel(pc\ pc')}{subtp((\theta_1 \rightarrow^{pc} \theta_2)\ (\theta_1' \rightarrow^{pc'} \theta_2'))} \text{ subtp/=>}$$

$$\frac{bl;\Gamma \vdash_{pc} e : \theta \quad subtau(\theta\ \theta') \quad sublevel(pc\ pc')}{bl;\Gamma \vdash_{pc'} e : \theta'} \text{ of/sub/ofe}$$

$$\frac{bl;\Gamma \vdash_{pc} c : \theta \quad subtau(\theta\ \theta') \quad sublevel(pc\ pc')}{bl;\Gamma \vdash_{pc'} c : \theta'} \text{ of/sub/ofc}$$

Explanation for expression type system:
For the security level labels of the type system we will follow the following principles

- High value is not allowed to be assigned to a low variable (explicit flow).
- Low assignment is not allowed inside a high guard (explicit flow).
- In case of a sequential evaluation of expression or running of commands, the blocking level of the latter must not be lower than the earlier commands or expressions (PSNI).
- A blocking command is rising the blocking level to its own to the level of its guard.

expressions: of/fn : the function is not actually evaluated at this stage and thus no restrictions. Assure that non of the levels of either the side effects (”LE”) or PC should change, the level of the function can always be considered low due to subtyping.

of/rec : a recursive function could run forever and thus if there is a high guard within a recursive function, it's blocking level will rise to that level. exmaple of a high blocking recursive function:

```
function HighBlock(h) {
  if h (HighBlock(h));
}
```

of/app: Takes an function as an expression and an expression as an input to that function. The label of the result can't be higher than the label of the function (as then the function would leak its result). And side effects of the function can't be lower than the label of the function(as it will leak the function) or the PC (explicit flow).

of/assign: Takes a property of an object and an expression and assigns the expression to that property. Assures that the level of variable expression is not lower than the PC and that the level of the value is not higher than the level of the variable. Assures that the level of the property is not higher than the level of it's object. e1 is evaluated before e2 and thus its bl can't be lower.

of/==: e1 is evaluated before e2 and thus its bl can't be lower. The level of the result is the highest of e1 and e2 so it doesn't leak any information about comparison of high value to a lower one.

Explanation for commands type system:
of/# : # is used to run an expression as a command and thus is not changing the expression by itself. Thus as long as the expression is well typed, the command is well typed.

of/var: Similar to of/assign, Assures that the level of the variable expression is not lower than the PC and that the level of the value property is not higher than the level of the variable. Assures that the level of the property is not higher than the level of it's object. e1 is evaluated before e2 and thus its bl can't be lower.

of/; : ; is the sequential command which allows to run a command after a certain command has terminated. Assuring that the type of the blocking level can only be higher in the latter command.

of/while: Takes expression as the condition and a command which will be run repeatedly as long as the condition holds. Assuring that the PC of the command is not

lower than the level of condition or its PC, assuring that the levels of the command is not higher than the level of the expression and assuring that the blocking level is risen to the level of the expression.

'while' example:

```
while (high){
    var low = 1;
};
outL(1);
```

This program will be rejected for 2 reasons: 1) the 'pc' level will not allow a low assignment inside a high guard loop. 2) the 'bl' level will not allow a low publication after a high guard loop.

of/if: Takes 3 arguments, an expression as the condition and 2 commands as the 'do' and 'else' clauses, in case the expression is true run c1 and else run c2. Assure that the PC of the commands are not lower than the level of condition or its PC, also assure that the levels of the commands are not higher than the level of the expression. Since the expression will be evaluated before the commands, it's blocking level is allowed to be lower.

'if' example:

```
if (high) {
    var low = 1;
}
```

This example will not pass thetype ystem, assignment will be forbidden by the pc due to implicit leak of the high guard to the low var.

of/skip: doesn't do anything and thus is always safe.

of/trycatchfinally: takes 4 arguemts: c1 as the command that runs as a try clause, θ as a type which is expected to be thrown and c2 as the catch clause and c3 as the finally clause. The pc2 will be the PC level of the highest possible throw PC of the try block, this works because the sub-typing of PC could always lower the PC once it is well typed. Thus the catch clause will run under the highest possible pc that could be in a throw command. example:

```
try {
    int low = 1;
    int high = 3;
    if (low == 3) {throw  11}
    else if (high == 3) {throw  13}
} catch(E) {int high2 = E}
finally{skip}
```

the initial PC level of C1 (i.e. the try block) might be low, the PC of C2 will be set to high due to the high guard, therefore the level of the throw PC could be at least as high, so the example will only pass the type judgment if the catch block will well type with a high PC (and thus also lower due to sub-typing). since c1 will always run before c2, and c3 will run after both c2 and c3, their bl levels are sorted alike, with c1 lower than c2, and c2 lower than c3.

of/throw: takes a new variable which it will be store its exception in, and an expression which will be the content thrown. assures that the type that its variable expects is the same as the type of the expression and that the pc level is maintained.

of/new: Assures that the argument is not higher than the object, e1 is evaluated before e2 and thus its bl is allowed to be lower.

of/field: Assures that the level of the final fieldlist will be as high of its highest property.

of/dot: assures that the level of the property refered will be as high as the level of the property.

of/newproperty: allow the level of the new property to be lower than the level of the continuation command, as the property by itself doesn't leak anything.

of/outL: Despite we didn't develop the output commands in the operational semantics yet, those are still defined in terms of the type system, as those will assure PSNI. A low output is only allowed in a low bl and low pc context as other ways it will progress leak or implicitly leak a secret. As well as that expression which is being low output must be low.

of/outH: Since high output is only observerable by a high observer and thus could be in any context and send any variable. It is also non blocking and thus is not increasing the bl level.

subtyping: The intuition to the subtyping rules for expressions and commands is as follows: The type itself should never change (thus a nat stays a nat). The labeled level of a type could always be safe to consider it higher (as it is more restrictive). In a case of a function, is that when the function is safe, and if the input is high it will safe also to consider the input as low, and if the output is low, it will also be safe to consider the output as high, if the side effects ("pc") were safe under high they will also be safe under low. The reason that there are no subtyping rules for the blocking level, is that a command which is safe under high bl, should force all the commands following it to be also safe under high, and therefor can not be considered as safe under low.

Note that in the implementation we make some restriction and permissiveness boundaries, as our system will be sound but not complete, meaning that not all safe programs will pass our judgment (but still all judgements are correct).

A restrictive consequence of our approach is that "dead code", despite being unreachable, will still be evaluated in the type system, as the type system will be eager and check all the code. For example:

```
try {
    int low = 3;
    int high = 3;
    if (low == 3) {throw  11}
    else if (high == 3) {throw  13}
```

```
} catch(E) {int low2 = E}
finally{skip}
```

the following code is demonstrating a try catch scenario, as will explained in the type system, the pc level of the catch clause will be determined by the highest pc level possible for a throw command, which in this case will be high as in the second 'if' case the throw will depend on the high guard. This code is secure despite the catch clause being low, because the throw with the high guard will never be reached, however the type system will check that code as well when deciding the minimum level of the pc of the catch clause and therefore will reject the low assignment in it.

In terms of PSNI permissiveness, low assignments after high level blocking command will be allowed as long as the variable will not be published.

7.2 Type System Implementation

Type system for IFC

```
objectable : tp.
fieldlist : tp.



<= : level → level → type.
<=/eq : <= L L.
<=/lt : <= low high.



join : level → level → level → type.
join/1 : join low low low.
join/2 : join high low high.
join/3 : join low high high.
join/4 : join high high high.


subtau : tau → tau → type.
subtp : tp → tp → type.
sublevel : level → level → type.
subtau/# : subtau (#' T L) (#' T' L')
      ← subtp T T'
      ← <' L L'.
sublevel/PC : sublevel PC PC'
      ← <' PC' PC.
subtp/nat: subtp nat nat.
subtp/bool: subtp bool bool.
```

```
subtp/one: subtp one one.

subtp/=>1 : subtp (=>' PC BL T1 T2) (=>' PC BL T1' T2)
      ← subtau T1' T1.
subtp/=>2 : subtp (=>' PC BL T1 T2) (=>' PC BL T1 T2')
      ← subtau T2 T2'.
subtp/=>3 : subtp (=>' PC BL T1 T2) (=>' PC' BL T1 T2)
      ← sublevel PC PC'.


ofe : level → level → exp → tau → type.
ofc : level → level → cmd → tau →  type.

off : property → tau → type.
offs : fields → tau → type.
ofk : level → cont → tau → tau → type.
ofo : obj → tau → type.

% ofe P BL E T   corresponds to:  PC; BL; G |- E T
% ofc PC BL C T   corresponds to: PC; BL; G |- C T
% off for properties.
% ofk PC K T1 T2  corresponds to: The catch clause expects
% something of type T1 and the continuation will yield
% something of type T2.
% ofo for objects.


```

<= stands for $\sqsubseteq$, meaning it takes 2 levels as arguments and accepts them if the level of the first argument is less or equal to the second argument. This has 2 cases in our case: either eq (eqaul) or lt (less than) when the first argument is low and the second high.

"join" stands for $\sqcup$, meaning it takes 3 levels as arguments and accepts them if the last argument is the atleast as high as the higher of the first and second argument. Naturaly it has 4 cases as can be seen in the code.

"subtau", "sublevel" and "subtp" are corresponding to their rules in the type system above.

"ofe" stands for a well typed expression. "ofc" stands for a well typed command. "off" stands for a well typed properties. "ofk" stands for a well typed continuations. "ofo" stands for a well typed objects. Recall that expressions and commands have 2 extra levels tracking implicit flows and progress. The continuation takes 2 levels (inside the taus) as it has to remember which level to expect and which level to return.

Rules for subtyping:

```
of/sub/ofe :  ofe PC' BL E T'
```

```
     ← subtau T T'
     ← sublevel PC PC'
     ← ofe PC BL E T.

of/sub/ofc :  ofc PC' BL C T'
     ← subtau T T'
     ← sublevel PC PC'
     ← ofc PC BL C T.
```

Corresponding to the subtyping rules.

Rules for expressions:

```
of/true : ofe PC BL true (#' bool L).

of/false : ofe PC BL false (#' bool L).

of/unit : ofe PC BL unit (#' one L).

of/z : ofe PC BL  z (#' nat L).

of/s : ofe PC BL (s E) (#' nat L)
  ← ofe PC BL E (#' nat L) .

of/fn : ofe PC BL (fn λx. C !x) (#' (=>' LE BL T1 T2) low)
          ← (Π x. ofe LE BL x T1 → ofc LE BL (C !x) T2).

of/app : ofe PC BL E1 (#' (=>' LE BL (#' T1 L1) (#' T2 L2)) L3)
    → ofe PC BL E2 (#' T1 L1)
    → <= L2 L3
    → join PC L3 PC2
    → <= PC2 LE
    → ofe PC BL (app E1 E2) (#' T2 L2).

of/undef : ofe PC BL undef (#' nat L).

of/assign : ofe PC BL2 (assign E1 F E2) (#' T1 L2)
       ← <= BL1 BL2
       ← <= PC L2
       ← <= L1 L2
       ← <= L3 L2
```

```
30        ← ofe PC BL1 E2 (#' T1 L1)
31        ← off F (#' T1 L2)
32        ← ofe PC BL2 E1 (#' objectable L3).
33
34
35  of/not : ofe PC BL (not E) (#' bool L)
36      ← ofe PC BL E (#' bool L).
37
38
39  of/== : Π T1. Π T2. ofe PC BL2 (== E1 E2) (#' bool L3)
40     ← ofe PC BL2 E2 (#' T2 L2)
41     ← ofe PC BL1 E1 (#' T1 L1)
42     ← join L1 L2 L3
43     ← <= BL1 BL2.
```

Correspoding to the expression rules. of/== has Π T1. ΠT2 in it first line (line 39) this is for the construction of proofs and has no logical implication. Some rules have those pis as well and could be ignored. Rules for commands:

```
1  % BL stands for Blocking Level in PSNI.
2
3  of/# :  ofe PC BL E (#' T L)
4       → ofc PC BL (# E) (#' T L).
5
6
7  of/var :  ofc PC BL2 (var E1 F (#' T1 L1) E2) (#' T2 L2)
8       ← <= BL1 BL2
9       ← <= L1 L3
10      ← (Π o. ofo o (#' objectable L1) →
11                          ofe PC BL2 (E2 !o) (#' T2 L2))
12      ← off F (#' T1 L1)
13      ← ofe PC BL1 E1 (#' objectable L3)
14      ← <= L2 L1
15      ← <= PC L1.
16
17
18 of/; : Π T1. ofc PC BL2 (; C1 C2) (#' T2 L4)
19   ← ofc PC BL2 C2 (#' T2 L3)
20   ← ofc PC BL1 C1 (#' T1 L2)
21   ← join L2 L3 L4
22   ← <= BL1 BL2.
23
24 of/while : ofc PC L1 (while E C) (#' T L3)
25    ← ofc L3 L1 C (#' T L4)
26    ← ofe PC L1 E (#' bool L2)
```

```
   ← join PC L2 L3
   ← <= L2 L1
   ← <= L2 L3.


of/if : ofc PC BL2 (if E C1 C2) (#' S L'')
   ← <= BL1 BL2
   ← join PC L L'
   ← ofe PC BL1 E1 (#' bool L)
   ← ofc L' BL2 C1 (#' S L'')
   ← ofc L' BL2 C2 (#' S L'')
   ← <= L'' L.

of/skip : ofc PC BL skip L.

of/trycatchfinally : Π PC'. ofc PC BL3
                (trycatchfinally C1 T1 C2 C3) T2
         ← <= PC PC'
         ← <= BL1 BL2
         ← <= BL2 BL3
         ← (Π k. ofk PC' k T1 T2 → ofc PC BL1 (C1 !k) T2)
         ← (Π e. ofe PC' BL e T1 → ofc PC' BL2 (C2 !e) T2)
         ← ofc PC BL3 C3 T2.

of/throw : Π T1. ofc PC' BL (throw K E) T
      ← ofk PC' K T1 T
      ← ofe PC' BL E T1.
```

Corresponding to the command rules.

Rules for Object system:

```
of/new : ofe PC BL2 (new E1 E2) (#' T L')
    ← <= BL1 BL2
    ← ofe PC BL1 E1 (#' objectable L)
    ← <= L' L
    ← ofe PC BL2 E2 (#' T L').

of/field : Π T. Π L1. Π L2. offs (field F FS) (#' fieldlist L3)
    ← off F (#' T L2)
    ← offs FS (#' fieldlist L1)
    ← join L1 L2 L3.
```

```
of/nil : offs nil (#' fieldlist low).

of/dot : ofe PC BL (dot E F) (#' T L)
  ← off F (#' T L)
  ← ofe PC BL E (#' objectable L).

of/newproperty : Π T'. ofc PC BL (newproperty λF. C !F) (#' T L)
  ← (Π F. off F (#' T' L') → ofc PC BL (C !F) (#' T L))
  ← <= L' L.




of/dec : ofe PC BL (dec E) (#' nat L)
  ← ofe PC BL E (#' nat L).

of/ref : ofe PC BL (ref O) (#' T L)
  ← ofo O (#' T L).

of/global : ofo global (#' objectable L).

of/emptyobj : ofe PC BL emptyobj (#' T low).

of/fix : ofe PC BL (fix λx. E !x) (#' (=>' LE BL2 T1 T2) low)
    ← (Π x. ofe LE BL x T1 → ofe LE BL (E !x) T2)
    ← join LE BL BL2.

```

Corresponding to the object system rules.

8 Soundness

Within this subsection, the soundness (in regard to security as the property of progress sensitive non-interference) of our type system will be proven. Due to lack of time, this section is lacking many of the formal proofs for the judgement rules. The lacking proofs should be done in a similar way to those provided. We also don't have the output commands outL(e) and outH(e) coded in our operational semantics, however for the sake of proving PSNI we will assume they exist (see rule of/outL in the type system).

PSNI definition:

Sets:

N : the set of names

R : the set of references

V : the set of values

O : the set of objects

L : set of levels

Semantic-objects:
ref : r
name : n
value (booleans, numbers, references) : v
object : o ::= $\{n_1, \ldots, n_k\} \mid \lambda$n.c
level : l ::= L | H

Non-referenced values are stored on the stack, referenced values on the heap.
stack S : N $\rightarrow$ V
heap H : R $\rightarrow$ O

Running programs perform I/O on memory: the stack and heap.

memory : $\mu ::= \langle S, H \rangle$
Running programs might perform outputs: output : ou : V
An output can be either an output to a low channel (”low output”) or to an high channel (”high output”). Recall that a program specifies the types of properties. From the program specification, one can therefore extract types along the following lines.
basic b^l : nat, bool, unit, ref
function $(bl, \theta_1 \rightarrow_{le} \theta_2)^l$
record : $\{\}^l$

From these types, one can construct a function, which maps each property, to its structural level, i.e. the information conveyed by knowing the value. i.e. the l in the types above.
structure Γ : N $\rightarrow L$
e.g. in a secret variable such as

```
1 var password = 123;
```

Γ(password) will return ”H” (high level), as it is a secret variable.

Two memories are l equivalent iff all structurally-l things on the stack are equal (i.e. memories agree on low bools, nats, etc), and all structurally-l things on the heap are equal (i.e. memories agree on low functions, and low records)

Definition of l-equivalence on memories:
$\langle S_1, H_1 \rangle \equiv_{Mem_l} \langle S_2, H_2 \rangle$ iff either of those hold:
1) $\forall n. S_1(n) \notin R \Rightarrow \Gamma(n) = l' \Rightarrow l' \sqsubseteq l \Rightarrow S_1(n) = S_2(n)$
2) $\forall n. S_1(n) \in R \Rightarrow \Gamma(n) = l' \Rightarrow l' \sqsubseteq l \Rightarrow H_1 \circ S_1(n) = H_2 \circ S_2(n)$
i.e. check if there is a reference in the memory and return its corresponding heap, compare the result; if they are of the same level of lower of the level you are trying to compare. Low output list ol : $\{$ ou$_1$, ... , ou$_k\}$
$\langle ol, c \rangle$: A configuration in our semantics (i.e. our semantic domain).

such that if command c has a low output, that output will be added to ol of the next command.
t : Trace (finite list of configurations)
$\langle \mu, c \rangle \Downarrow$ t : Transition system q emits trace t.

2 output lists are list low-equivalent if all the items of each list of the are equal, and the lists are of the same length.
Thus the following relation $ol \equiv_{List_L} ol'$ will be defined by the following rules :

$$\frac{}{e \equiv_{List_L} e} \; ol/empty$$

$$\frac{ou \neq hd(ol) \quad ou' \neq hd(ol') \quad ou \equiv ou' \quad ol \equiv_{List_L} ol'}{ou \bullet ol \equiv_{List_L} ou' \bullet ol'} \; ol/match$$

Thus matching comparison of 2 lists will be done by traversing and low comparing each item of the ol list until reaching that both are empty. $ou \neq hd(ol)$ checks that the list is not empty (as the 2 lists have to be of the same length). "e" is the empty list. $\bullet$ is a cons operation of lists (thus the left hand is the first item and the right hand is the rest of the list) .

2 sequences of output lists traces are progress-sensitive indistinguishable if the one has no ability to diverge terminate prior to the other. Thus if a sequence is diverging, the other sequence must diverge at the same point of progress. This could be assured through checking the low equivalence of the output lists of each configuration of a trace.

$t_1 \equiv_{PS_L} t_2$:

$$\frac{}{e \equiv_{PS_L} e} \; t/empty$$

$$\frac{p_1 \equiv_{List_L} p_2 \quad p_2 \bullet s \equiv_{PS_L} t}{p_1 \bullet p_2 \bullet s \equiv_{PS_L} t} \; t/left$$

$$\frac{q_1 \equiv_{List_L} q_2 \quad s \equiv_{PS_L} q_2 \bullet t}{s \equiv_{PS_L} q_1 \bullet q_2 \bullet t} \; t/right$$

$$\frac{p \neq hd(s) \quad q \neq hd(t) \quad q \equiv_{List_L} p \quad s \equiv_{PS_L} t}{p \bullet s \equiv_{PS_L} q \bullet t} \; t/match$$

The intuition is similar to the matching of output lists, besides that here it is allowed for any 2 following configurations to have low equal output lists (because 2 programs which one of them has extra commands that don't change the low output list are still low PS equal). "q" and "p" are ol parts of a configuration (note that the command part of the configuration is ignored, as 2 programs are low-PS equal even if they run different commands, as long as they have the same list of outputs at each progress point), "s" and "t" are output traces. And thus the definition of PSNI(c) in our language will be:

$PSNI(c) = \forall \mu_1, \mu_2 . \mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle c, \mu_1 \rangle \Downarrow t_1 \Longrightarrow \exists t_2 . \langle c, \mu_2 \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$

Lemma (H pc) :

$\langle \mu, c \rangle$: A memory configuration in our semantics .
mt : Memory trace (finite list of memory configurations)
q $\Downarrow$ mt : Transition system q emits memory trace mt.
mt $\equiv_{MT_L} mt$:

$$\frac{}{e \equiv_{MT_L} e} \; mt/empty$$

$$\frac{p_1 \equiv_{Mem_L} p_2 \quad p_2 \bullet s \equiv_{MT_L} t}{p_1 \bullet p_2 \bullet s \equiv_{MT_L} t} \; mt/left$$

$$\frac{q_1 \equiv_{ML_L} q_2 \quad s \equiv_{MT_L} q_2 \bullet t}{s \equiv_{MT_L} q_1 \bullet q_2 \bullet t} \; mt/right$$

$$\frac{p \neq hd(s) \quad q \neq hd(t) \quad q \equiv_{Mem_L} p \quad s \equiv_{MT_L} t}{p \bullet s \equiv_{MT_L} q \bullet t} \; mt/match$$

The logic in these matching is similar to PS matching, besides that here "q" and "p" are states (memory part of a memory configuration) and "t" and "s" are memory traces.

$bl; \Gamma \vdash_{pc} c : T$
AND
pc = High
$\Rightarrow$
$\forall m, m'$
$\langle c, m \rangle \Downarrow mt \longrightarrow \langle skip, m' \rangle$
$\Rightarrow$
$m \equiv_{Mem_L} m'$
AND
$mt \equiv_{MT_L} []$

Meaning that if a command is well typed under a High pc level, in the case where the command c emits a trace which has no low configuration, for all memories m and m' such that m is the memory prior to the run of the command and m' is the memory after the run of the command, all the Low parts of m will be equal to m'.
$\langle c, m \rangle \Downarrow tl \longrightarrow \langle c', m' \rangle$
tm ::= [tl]
$\langle c, m \rangle \Downarrow tm \longrightarrow \langle skip, m' \rangle$
$\Rightarrow$
$\exists c_1, c_2$ *and* m_1, m_2 *and* $tl_1, tl_2 ...$
s.t.
$\langle c, m \rangle \Downarrow tl_1 \rightarrow \langle c_1, m_1 \rangle \Downarrow tl_2 \rightarrow \langle c_2, m_2 \rangle ... \langle skip, m' \rangle$

meaning that if the trace isn't empty of low commands, the initial memory of m and the memory m' after running the command will still be low equal (because there will be no low assignments). (Until here the High pc lemma)

$\Gamma = \cdot | e | f | o | c | k$
security definition (PSNI):

PSNI (c) :
$\forall \mu_1, \mu_2 . \mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle \mu_1, c \rangle \Downarrow t_1 \implies \exists t_2 . \langle \mu_2, c \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$
PSNI (e) :
$\forall \mu_1, \mu_2 . \mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle \mu_1, e \rangle \Downarrow t_1 \implies \exists t_2 . \langle \mu_2, e \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$
whereμ is memory and t is an output list trace.
Lemma pc for e or c:
$(bl; \Gamma \vdash_{pc} e : \theta) \Rightarrow \forall l . l \sqsubseteq pc \Rightarrow (bl; \Gamma \vdash_{l} e : \theta)$
Lemma $T^l(\theta)$ for e or c:
$(bl; \Gamma \vdash_{pc} e : T^l) \Rightarrow \forall l' . l \sqsubseteq l' \Rightarrow (bl; \Gamma \vdash_{pc} e : T^{l'})$
Der *Derivation*
$\mathscr{D} : Der$
Theorm:$(bl; \Gamma \vdash_{pc} e : \theta) \Rightarrow PSNI(e)$
proof :
We prove:
$\forall \mathscr{D} . \forall e, \Gamma, pc, l, bl . \mathscr{D} :: (bl; \Gamma \vdash_{pc} e : T^l) \Rightarrow PSNI(e)$ (X)
This proves our theorem, since:
$(bl; \Gamma \vdash_{pc} e : T^l) \iff \exists \mathscr{D} . (bl; \Gamma \vdash_{pc} e : T^l)$
Induction in the structure of $\mathscr{D}$
Base cases:
of/z:
$case\ z := \dfrac{}{bl; \Gamma \vdash_{pc} z : nat^l}$ of/z
By (of/z):
$e = z$ (e)
To prove:
PSNI(e).
i.e. PSNI (e) :
$\forall \mu_1, \mu_2 . \mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle \mu_1, e \rangle \Downarrow t_1 \implies \exists t_2 . \langle \mu_2, e \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$

Pick
μ_1, μ_2
s.t.
$\mu_1 \equiv_{Mem_L} \mu_2$
Pickt_1, t_2 $(\equiv)$
s.t.
$\langle \mu_1, e \rangle \Downarrow t_1$ and $\langle \mu_2, e \rangle \Downarrow t_2$.
To Prove:
$t_1 \equiv_{PS_L} t_2$
By (e), we have that :
$\langle \mu_1, e \rangle \Downarrow \langle [], e \rangle = t_1$ and $\langle \mu_2, e \rangle \Downarrow \langle [], e \rangle = t_2$.(becuase (e) doesn't output anything) $(\Downarrow)$
By ($\Downarrow$), we get:
$t_1 \equiv_{PS_L} t_2$. $(\equiv \varsigma)$
Thus
PSNI(e), note that all axioms should be proven in the same way (axiom, of/true, of/false, of/unit, of/undef, of/skip, of/nil, of/outH)

Case nat $\overline{n}$ of/s : $\dfrac{\mathscr{D}' :: bl; \Gamma \vdash_{pc} e' : nat^l}{bl; \Gamma \vdash_{pc} (s\ e') : nat^l}$ of/s
Inductive step:
Assume: (X) holds for all structurally smaller $\mathscr{D}$' (IH)
Case (of/s)
Pick
e, Γ, bl, pc, l
s.t.
$\mathscr{D} :: (bl; \Gamma \vdash_{pc} e : nat^l)$ $(\vdash)$
By (of/s),
$e = (s\ e')$ (e)
By ($\vdash$) and (e), we have, for some $\mathscr{D}$' that
$\mathscr{D}' :: (bl, \Gamma \vdash_{pc} e' : nat^l)$.
By (IH), since $\mathscr{D}$' is structurally smaller than $\mathscr{D}$, we have PSNI(e') (e')
To prove:
$PSNI(e)$
$i.e. \forall \mu_1, \mu_2 . \mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle \mu_1, e \rangle \Downarrow t_1 \implies \exists t_2 . \langle \mu_2, e \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$
Pick
μ_1, μ_2
s.t.
$\mu_1 \equiv_{Mem_L} \mu_2$
Pick t_1, t_2 $(\equiv)$
s.t.
$\langle \mu_1, e \rangle \Downarrow t_1$ and $\langle \mu_2, e \rangle \Downarrow t_2$.
To Prove:
$t_1 \equiv_{PS_L} t_2$

By (e'), we have that :
$\forall t_1'.\langle \mu_1, e' \rangle \Downarrow t_1' \implies \exists t_2'.\langle \mu_2, e' \rangle \Downarrow t_2' \wedge t_1' \equiv_{PS_L} t_2'$ (t')
Since s doesn't output anything, we get that the list of outputs of e' is equal to e (i.e. (s e'))
pick any ol, tt, tt', s.t. :
$t_1' = tt \bullet \langle e', ol \rangle \wedge t_2' = tt' \bullet \langle e', ol \rangle \implies t_1 = t_1' \bullet \langle e, () \bullet ol \rangle \wedge t_2 = t_2' \bullet \langle e, () \bullet ol \rangle$ (t)
Because:$() \bullet ol \equiv_{List_L} () \bullet ol$ we get, according to definition of low-PS equivalence, that:
$t_1 \equiv_{PS_L} t_2$
Thus:$PSNI(e)$ note that all rules with only 1 assumption should be proven in the same
way (of/dec, of/fn, of/fix, of/not, of/ref, of/# ,of/return) besides of/outL , since it has a low output.
(of/if)
$case\ if :=$

$$\frac{\mathscr{D}_1 :: bl_1; \Gamma \vdash_{pc} e : bool^{l_1} \quad \mathscr{D}_2 :: bl_2; \Gamma \vdash_{pc \sqcup l_1} c_1 : T^{l_2} \quad \mathscr{D}_3 :: bl_2; \Gamma \vdash_{pc \sqcup l_1} c_2 : T^{l_2} \quad \gamma \vdash l_2 \sqsubseteq l_1 \quad \gamma \vdash bl_1 \sqsubseteq bl_2}{bl_2; \Gamma \vdash_{pc} if(e)\ \{c_1\}\ else\ \{c_2\} : T^{l_2}}\ \text{of/if}$$

$\mathscr{D}_4$ and $\mathscr{D}_5$ are the following (couldn't write that in the rule due to lack of space):
$\mathscr{D}_4 :: \gamma \vdash l_2 \sqsubseteq l_1 \quad \mathscr{D}_5 :: \gamma \vdash bl_1 \sqsubseteq bl_2$
Case c:
Inductive step:
Assume: (X) holds for all structurally smaller $\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3$ and that $\mathscr{D}_4, \mathscr{D}_5$ also hold (IH)
Pick:
e, c_1, c_2, T,Γ, l_2, pc, bl_2
s.t.
$\mathscr{D} :: bl_2; \Gamma \vdash_{pc} if(e)\ \{c_1\}\ else\ \{c_2\} : T^{l_2}$ $(\vdash)$
By (of/if),
$c = if(e)\ \{c_1\}\ else\ \{c_2\}$ (c)
By $(\vdash)$ and (c), we have, for some $\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3, \mathscr{D}_4, \mathscr{D}_5, e, c_1, c_2, T, \Gamma, l_1, l_2, pc, bl_1, bl_2$ such that:
$\mathscr{D}_1 :: bl_1; \Gamma \vdash_{pc} e : bool^{l_1} \quad \mathscr{D}_2 :: bl_2; \Gamma \vdash_{pc \sqcup l_1} c_1 : T^{l_2} \quad \mathscr{D}_3 :: bl_2; \Gamma \vdash_{pc \sqcup l_1} c_2 : T^{l_2}$
$\mathscr{D}_4 :: \gamma \vdash l_2 \sqsubseteq l_1 \quad \mathscr{D}_5 :: \gamma \vdash bl_1 \sqsubseteq bl_2$
By (IH), since $\mathscr{D}_1, \mathscr{D}_2, \mathscr{D}_3, \mathscr{D}_4, \mathscr{D}_5$ are structurally smaller than $\mathscr{D}$, we
have PSNI(e), PSNI(c_1), PSNI(c_2) (PS)
To prove:
$PSNI(c)$
$i.e. \forall \mu_1, \mu_2.\mu_1 \equiv_{Mem_L} \mu_2 \wedge \langle \mu_1, e \rangle \Downarrow t_1 \implies \exists t_2.\langle \mu_2, e \rangle \Downarrow t_2 \wedge t_1 \equiv_{PS_L} t_2$
Pick
μ_1, μ_2
s.t.
$\mu_1 \equiv_{Mem_L} \mu_2$
Pick t_1, t_2 $(\equiv)$
s.t.
$\langle \mu_1, e \rangle \Downarrow t_1$ and $\langle \mu_2, e \rangle \Downarrow t_2$.
To Prove:
$t_1 \equiv_{PS_L} t_2$

By (c) and $(\equiv)$, we have that :
$\langle \mu_1, (if\ (e)\{c_1\}else\{c_2\}) \rangle \Downarrow t_1 \wedge \langle \mu_2, (if\ (e)\{c_1\}else\{c_2\}) \rangle \Downarrow t_2$

By the operational semantics (run/if), we have that c (i.e. if(e, c_1, c_2)) will either run c_1 or c_2 if e is low and μ_1 and μ_2 are low memory equal, then e will always return the same result for both memories (due to (PS)) and because c_1 and c_2 are also PSNI secure, whichever of them that will run will produce the same output under different memories, and hence if e is low, c will be PSNI, in the case that l is high, it could be that both c_1 and c_2 will run under 2 different low memory equal memories (as their high parts are different) however this should not affect the list of low output that were sent, as low output is not allowed under high pc level, and high pc level is enforced on both c_1 and c_2 if l is high. So c will be PSNI also in the case in which e is high.
Rising the pc level is the level of e is high will also assure no implicit leak and will hold the high pc lemma, as low assignments are not allowed under low pc (pc)
Progress sensitive is relevant in this command, as e will always be evaluated before c_1 or c_2 will run, and therefore the blocking level of e is allowed to be lower than the blocking level of c_1 and c_2 includes any sequential operations (bl)
By $(\equiv)$, (ps) , (pc) and (bl), we get :
$t_1 \equiv_{PS_L} t_2$
Thus:$PSNI(c)$

Similar proofs should be constructed for all the commands, following the PSNI definition.

9 Implementation

9.1 Type Derivation Examples:

We will present the type derivation for the following JavaScript program:

```
var safe = {data:3,
    show: function(){
        return this.data + 1;
    },
};
safe.cipher = 2;
safe.decrypt = function() {return this.cipher - 1};
safe.encrypt = function() {this.cipher = 1 + this.data};
safe.encrypt();
return safe.decrypt();
```

this program will be written as the following in our language:

```
advobject2 : cmd
  = decf \!data. decf \!show.
    const !(#' objectable low)
      !(field data (s (s (s z)))
       (method show (\!this. fn \!x.
                return (s (deref (doto !this data))))
       nil))
```

```
!(\!safe.

decf \!cipher. decf \!encrypt. decf \!decrypt.

; (addfield safe cipher (s (s z)))

(; (addmethod safe encrypt (\!this. fn \!x.
   ; (# (assign (doto this cipher)
          (s (deref (doto this data)))))
   (return unit)))

(; (addmethod safe decrypt (\!this. fn \!x.
   return (dec (deref (doto !this cipher)))))

(; (# (app (deref (dot safe encrypt)) unit))

(return (app (deref (dot safe decrypt)) z)))))).
```

using this program we could construct the following derivation:

```
deriv_advobject:
   ofc low low advobject2 (#' nat low) =
of/decf (#' nat low ) (\!data. \!datap.
of/decf (#' (=>' low (#' nat low) (#' nat low)) low) ((\!show. \!showp.
of/var  (\!safe. \!safep. (of/;  objectable <=/eq join/1
  (of/# low (of/assign (of/record (of/field (nat) low low low join/1 <=/eq (of/s (of/s
   (of/s (of/z : ofe low z (#' nat low)))))
  (of/method (=>' low (#' nat low) (#' nat low)) low low low join/1 <=/eq
                                        (\!this. \!thisp. of/fn low low (\!xx. \!xxx.
                                 (of/return (of/s (of/deref (of/doto thisp datap))))))
                                             (of/nil : offs nil (#' fieldlist low) )
    (showp)) datap ))
<=/eq <=/eq safep)) \!safe. \!safep.
( of/decf (#' nat low) (\!cipher. \!cipherp.
of/decf (#' (=>' low (#' one low) (#' one low)) low) (\!encrypt. \!encryptp.
of/decf (#' (=>' low (#' nat low) (#' nat low)) low) (\!decrypt. \!decryptp.
(of/; nat <=/eq join/1 (of/addfield low low <=/eq <=/eq (of/s (of/s
(of/z : ofe low z (#' nat low))))
  cipherp
  safep)

(of/;   (=>' low (#' one low) (#' one low))  <=/eq join/1 (of/addmethod low
low <=/eq <=/eq
(\!this. \!thisp. of/fn low low (\!xx. \!xxx.
```

```
( of/;  nat <=/eq join/1 (of/# low (of/assign (of/s (of/deref (of/doto
thisp cipherp)))
<=/eq <=/eq  (of/doto thisp datap)))
(of/return (of/unit: ofe low unit (#' one low))))))
encryptp
safep)

(of/;   (=>' low (#' nat low) (#' nat low))  <=/eq join/1
(of/addmethod low low <=/eq <=/eq
(\!this. \!thisp.  of/fn low low (\!xx. \!xxx.
( of/return (of/dec (of/deref (of/doto thisp cipherp))))))
decryptp
safep)

(of/;  one <=/eq join/1 (of/# low (of/app (of/deref (of/dot safep encryptp))
(of/unit: ofe low unit (#' one low)) <=/eq join/1 <=/eq ))
(of/return (of/app (of/deref (of/dot safep decryptp))
(of/z: ofe low z (#' nat low)) <=/eq join/1 <=/eq ) ))))))))))))).
```

which means that the program 'advobject2' is well typed under the type nat^{Low} low pc and low bl.

The following example is a simple program which assigns private information and public information properly.

```
var t; // high
var l; // low
var h; // high
l = 1;
t = 0;
while (h == 1) {t = 1}
while (!(t == 1)) { t = 1}
l = 0;
```

corresponds to celf code:

```
advwhile3 : cmd =
var (#' nat high) \!t.
var (#' nat low) \!l.
var (#' nat high) \!h.
; (# (assign l (s z)))
( ; (# (assign t z))
( ; (while (== (deref h) (s z))
             (# (assign t (s z))))
( ; (while (not (== (deref t) (s z)))
            (# (assign t (s z))))
```

```
( ; (# (assign l z))
skip))))).
```

with derivation:

```
deriv_advwhile2:
   ofc low high advwhile3 (#' one high) =
       of/var (\!t. \!tt.
       of/var (\!l. \!ll.
       of/var (\!h. \!hh.
       of/; nat <=/lt join/3 (of/# (of/assign (of/s (of/z : ofe low low z
       (#' nat low))) <=/eq <=/eq ll))
       (of/; nat <=/lt join/4 (of/# (of/assign  (of/z : ofe low low z
       (#' nat high)) <=/eq <=/lt tt))
       (of/; nat <=/eq join/4 (of/while <=/eq <=/eq (of/# (of/assign
       (of/s (of/z : ofe high high z (#' nat high))) <=/eq <=/eq tt))

       (of/== nat nat join/4 (of/s (of/z : ofe low high z
       (#' nat high))) (of/deref hh))  join/3)
       (of/; nat <=/eq join/4 (of/while <=/eq <=/eq (of/# (of/assign
       (of/s (of/z : ofe high high z (#' nat high))) <=/eq <=/eq tt))

         (of/not (of/== nat nat join/4 (of/s (of/z : ofe
         low high z (#' nat high))) (of/deref tt)))
 join/3)

(of/; nat <=/eq join/3 (of/# (of/assign  (of/z : ofe low
    high z (#' nat low)) <=/eq <=/eq ll))
(of/skip: ofc low high skip (#' one high))))))))).
```

note that the command is only well typed with the bl level being ”high”, because it includes a while loop with a high guard.

Part IV
Closing

10 Related Work

10.1 Formal Semantics

In the paper (Meffeis et al, 2008) Formal semantics of JS are presented, those semantics are very detailed with many operational complexity. These details are too complex for the scope of our intentions, as they do not change the impact of the IFC type systems. f.ex. : arrays are a separate expression types despite that for IFC purpose they could just be considered as objects.

10.2 Information-Flow Control for JavaScript

In this section the state of art IFC enforcement for JS will be presented. This will be done in order to show to context and relevance of our work in this field.

10.2.1 JSFlow

JSFlow (Hedin, 2014) is a tool which enforces IFC on JS program during run-time, this is done due to dynamically enforce run-time monitor on JS. This approach is specially useful due to its ability to enforce NI also on dynamic code which is provided to the program during run-time (in the ”eval” command). In addition this approach could also enforce NI on external libraries provided by APIs.

10.2.2 Multi-Execution (FlowFox, Facets)

Shown in (Schmitz et al, 2018) is a tool for IFC enforcement by a synthesis of Multiple Facets (MF) and Secure Multi-Execution (SME), this approach is running many copies of a program while carefully adapting their semantics in order to avoid leakage. The advantages of this approach is that it guarantees TSNI, and transparency (it does not deny the behavior of safe programs regardless of the syntax, thus reducing false alarms). Disadvantage might be the overhead and resource requirement for such dynamic monitoring.

10.2.3 For Chromium

Approach presented by (Bauere et al, 2015) is a Run-Time Monitor for chromium (an open source browser which is the base of various popular browsers such as Google Chrome and Microsoft Edge). This approach also supports rich policy specification such as same origin policy (SOP), content security policies (CSPs).

10.2.4 WebKit

Dynmaic enforcement run-time monitor in the bytecode of js, for WebKit(the JS engine used in Safari and other open-source browsers). The advantage of monitoring the bytecode is the reduced overhead. (Rajani et al, 2014)

11 Conclusion

We have demonstrated that substrucal logics can be used to enforce IFC, we have implemented IFC rules in a type system which fits with JavaScript Core's syntax. We provided a logical derivation for such example, statically proving PSNI properties of a JavaScript Core program.

Thus we have shown a proof of concept for the possibility of statically proving IFC properties on JavaScript program, this could be developed to cope with the state of art problems with information leakage in the area of dynamic languages.

12 Reflection

The construction of the type system and operational semantics went well. I find the topics interesting and in addition, I had already made a preparation project that was related to these topics. Constructing definitions and proofs were more difficult. It is a complex issue that I found difficult to learn from the literature, especially since our case in JSL is unique, which required me to acquire in depth understanding of the process of constructing PSNI proofs.

13 Future Work

My suggestion for future work includes the following:

- Finish the soundness (I intend to attempt achieving this and present at the oral exam).
- More levels to the lattice could be added.
- The commands for output, namely outl() and outH(), could be added.
- A compiler could be made to JSL so it could run faster for more realistic use.
- Make JSL in a way that it could type check automatically (thus make type derivations automatically).
- Make sugaring of the more common used libraries of JS and ecma standards.
- Add more data structures such as lists, arrays and strings.
- Fix bugs in the types of the type system e.g. in of/if : types of clauses should be allowed to differ and of/assign should allow to assign a different type to a variable from what it was before.

14 Reference:

2020 Developer Survey [https://insights.stackoverflow.com/survey/2020] accessed [27/08/2020]

Askarov, A. and Sabelfeld, A., 2009, July. Tight enforcement of informationrelease policies for dynamic languages. In 2009 22nd IEEE Computer Security Foundations Symposium (pp. 43-59). IEEE.

Askarov, A., Hunt, S., Sabelfeld, A. and Sands, D., 2008, October. Termination-insensitive non-interference leaks more than just a bit. In European symposium on research in computer security (pp. 333-348). Springer, Berlin, Heidelberg.

Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M. and Tian, Y., 2015, February. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In NDSS. Bichhawat, A., Rajani, V., Garg, D. and Hammer, C., 2014, April. Information flow control in WebKit's JavaScript bytecode. In International Conference on Principles of Security and Trust (pp. 159-178). Springer, Berlin, Heidelberg.

Church, A., 1985. The calculi of lambda-conversion (No. 6). Princeton University Press.

ECMAScript language specification, 1999 - ecma Guha, A., Saftoiu, C. and Krishnamurthi, S., 2010, June. The essence of JavaScript. In European conference on Object-oriented programming (pp. 126-150). Springer, Berlin, Heidelberg.

Harper, R., Honsell, F. and Plotkin, G., 1993. A framework for defining logics.

Hedin, D. and Sabelfeld, A., 2012. A Perspective on Information-Flow Control. Software safety and security, 33, pp.319-347.

Hedin, D., Birgisson, A., Bello, L. and Sabelfeld, A., 2014, March. JSFlow: Tracking information flow in JavaScript and its APIs. In Proceedings of the 29th Annual ACM Symposium on Applied Computing (pp. 1663-1671). Int'l Standard, I.S.O., 1999. IEC 9899: 1999 (E). Programming Languages—C, ISO/IEC.

Journal of the ACM (JACM), 40(1), pp.143-184. Maffeis, S., Mitchell, J.C. and Taly, A., 2008, December. An operational semantics for JavaScript. In Asian Symposium on Programming Languages and Systems (pp. 307-325). Springer, Berlin, Heidelberg.

Moore, S., Askarov, A. and Chong, S., 2012, October. Precise enforcement of progress-sensitive security. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 881-893).

Pfenning, F. and Schurmann, C., 1999, July. System description: Twelf—a meta-¨ logical framework for deductive systems. In International Conference on Automated Deduction (pp. 202-206). Springer, Berlin, Heidelberg.

Pfenning, F., 1992. Computation and deduction. Unpublished lecture notes. Pfenning, F. and Elliott, C., 1988, June. Higher-order abstract syntax. In ACM sigplan notices (Vol. 23, No. 7, pp. 199-208). ACM.

Resources for developers, by developers. [https://developer.mozilla.org/en-US/docs/Web/JavaScript]. Accessed [27/08/2020]

Sabelfeld, A. and Myers, A.C., 2003. Language-based information-flow security. IEEE Journal on selected areas in communications, 21(1), pp.5-19.

Schack-Nielsen, A. and Schurmann, C., 2008, August. Celf–A logical framework¨ for deductive and concurrent systems (System description). In International Joint Conference on Automated Reasoning (pp. 320-326). Springer, Berlin, Heidelberg.

Schack-Nielsen, A. and Schürmann, C., 2008, August. Celf–A logical framework for deductive and concurrent systems (System description). In International Joint Conference on Automated Reasoning (pp. 320-326). Springer, Berlin, Heidelberg.

Schmitz, T., Algehed, M., Flanagan, C. and Russo, A., 2018, January. Faceted secure multi execution. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (pp. 1617-1634).