Investigating the Benefits of Ownership in Static Information Flow Security

Morten Meyer Rasmussen & Nikolaj Lepka

May 29, 2019

Abstract

This report is an exploration of the effects of applying a security monad in a language that supports Rust-style ownership. For this, we have designed a simple *While Language* to work on and to make examples in. We have extended the While Language with the ownership model found in Rust to see what guarantees this grants us in relation to a program's security and how it helps to avoid leaking secret information. From there we have looked at some of the issues with ownership and tried to fix them with using a security monad. The monad helps with some of the issues with ownership, because it could encapsulate the essential parts of the program and the secret information so it becomes more difficult to leak.

Contents

1 Introduction

2015 saw the birth of a new programming language called *Rust*. Rust is rather unique as a programming language, bucking trends by being a high-level language that focuses on *zero cost abstractions* without the use of a garbage collector, not unlike C++ before it[KN18]. Unlike C++ however, Rust has a special way of handling variables, commonly known as the *ownership model*.

In short, the ownership model is a static analysis tool built into the compiler, which assigns the ownership of data to variables, keeping track of how this data is used and moved. Once moved, ownership of the data is transferred, and the variable of origin becomes unusable unless assigned anew. This introduces some special cases for the programmer, such as variables unexpectedly getting moved, or references being used unsafely, forcing the programmer to rethink their approach.

```
// Short example
let a = MyStruct::new();
let b = a;
println!("{}", a); // Error, a was moved prior to use
```

The reference system works on a *single source of truth* principle, which ensures that only one reference with mutable access can exist to a variable at any given time. This aims to prevent a whole class of memory errors that could otherwise happen if no proper precautions were taken. Errors which include use-after-free, and double-free.

In previous work, we worked on taint analysis and a security library for Haskell which we implemented into Rust[RL19]. We can compare the Rust ownership model to that security library and see if it makes any difference in regard to the security properties of a program.

In this report, we investigate the ownership model to see if it has any impact on the creation of sofware with side-effects. We will also implement a security monad to simulate what we did in [RL19], while still working within the side-effect-ful limitations of Rust. We do this by creating a small toy language called *Sec While*, which aims to encapsulate the security monad in conjunction with ownership.

The report is split into three stages: First we define a basic *While* language, then we extend it to include ownership, then lastly we add the monadic capabilities. Along the way we discuss the merits of these additions, and why we believe them to be important.

We will in every stage of the report use some examples to discuss the progress of securing a programing language so the user not unintended leak secure information. An example could be how we validate user information.

```
let user_info = input;
user_info = sanitize(user_info);
let result = "";
let access_key = crypto(user_info);
let access_token = lookup(access_key);

if (access_token == "ok")
    result = "access granted";
else
    result = "access denied";

output = result;
```

In the following sections we will try to develop the example to become more secure so that a leak is more difficult to occur.

2 While

To start out we will define a little *While* language. The purpose of this language is to have a basic language to later extend to model Rust's ownership memory model in a small enclosed and easy to control environment, such that we do not have to deal with the entirety of a modern programming language's standard library and features otherwise irrelevant for the task at hand. The goal of While is to encapsulate just enough of Rust's features to be able to show that static information flow security benefits from the ownership model. While the language is not made to look like Rust in any significant

way, many of the decisions do reflect Rust's language structure as a whole. The most significant departure from Rust's core syntax comes in the form of our `let` bindings (Section 2.6) and our references (Section 3).

2.1 Syntax

The syntax of While is loosely based on other existing toy languages, but with an explicit notion of scopes. The purpose of scopes is two-fold: they both serve as a natural grouping of a sequence of statements, while also providing a means to enforce the ownership rules and variable destruction, as variables get cleaned up at the end of a scope. Ownership along with *unique* and *shared* references will be added later in Section 3.

$$
\begin{aligned}
main &::= stmt \\
stmt \in Stmt &::= \texttt{while}\ expr\ \texttt{do}\ stmt && \text{(While-Loop)} \\
&\mid \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt && \text{(If-Statement)} \\
&\mid \texttt{let}\ x : type = expr\ \texttt{in}\ \{stmt\} && \text{(Let)} \\
&\mid x = expr && \text{(Assignment)} \\
&\mid stmt; stmt && \text{(Composition)} \\
&\mid \texttt{skip} && \text{(Skip)} \\
expr \in Expr &::= x && \text{(Variables)} \\
&\mid expr\ \Diamond_b\ expr && \text{(Binary Operations)} \\
&\mid \Diamond_u\ expr && \text{(Unary Operations)} \\
&\mid (expr, expr) && \text{(Tuple)} \\
&\mid \texttt{first}(e) && \text{(Tuple First)} \\
&\mid \texttt{second}(e) && \text{(Tuple Second)} \\
&\mid v && \text{(Values)} \\
type \in Type &::= \texttt{int} && \text{(Integers)} \\
&\mid \texttt{str} && \text{(Strings)} \\
&\mid (type, type) && \text{(Pairs)} \\
x \in Var &::= \text{variable names} \\
v \in Val &::= \text{values}
\end{aligned}
$$

There is no direct syntax for handling I/O instead, if needed, it is provided by adding two predefined variables *input* and *output* to the environment. We will also be using functions calls in our examples as needed when it makes sense to do so.

2.2 Sets & Functions Used in the Semantics

Variables Var is the infinite set of variable names permitted in the language. Examples include `x`, `y`, `public`, etc.

Values

$$Val = \mathbb{Z} \cup String \cup (Val \times Val)$$

Is the set of legal values in the language. These values reflect the instances of the types defined in the syntax.

Types $Type$ is the set of types permitted in While. Recall the definition from Section 2.1:

$$type \in Type ::= \texttt{int} \mid \texttt{str} \mid (type, type)$$

The language consists of three basic types: integers, strings, and tuples. The purpose of these is to show how different types can have a different effect on the semantics. In Section 3.3 we define the notion of a *Copy* subtype, which alters the semantics of the types within it.

Environment We can then define the environment Σ as

$$\Sigma = Var \to Val$$

where each input variable points to an instance.

As an example, $\sigma = \{x \mapsto 42, y \mapsto \text{"}hello\text{"}\}$ is a valid environment as per our definition.

2.3 Statement Semantics

In this section we shortly go over the various statement semantics, to showcase how the language behaves. These semantics do not take the types of expressions into consideration, that is all relegated to the type system in Section 2.5, where γ is also defined.

$$\text{(While True)}\ \frac{\gamma \vdash \langle S, \sigma\rangle \to \sigma'' \quad \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma''\rangle \to \sigma' \quad \gamma, \sigma \vdash e \to 1}{\gamma \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma\rangle \to \sigma'}$$

$$\text{(While False)}\ \frac{\gamma, \sigma \vdash e \to 0}{\gamma \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \sigma\rangle \to \sigma}$$

$$\text{(If True)}\ \frac{\gamma \vdash \langle S_1, \sigma\rangle \to \sigma' \quad \gamma, \sigma \vdash e \to 1}{\gamma \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \sigma'}$$

$$\text{(If False)}\ \frac{\gamma \vdash \langle S_2, \sigma\rangle \to \sigma' \quad \gamma, \sigma \vdash e \to 0}{\gamma \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \sigma'}$$

$$\text{(Skip)}\ \frac{}{\gamma \vdash \langle \texttt{skip}, \sigma\rangle \to \sigma}$$

$$\text{(Compose)}\ \frac{\gamma \vdash \langle S_1, \sigma\rangle \to \sigma'' \quad \gamma \vdash \langle S_2, \sigma''\rangle \to \sigma'}{\gamma \vdash \langle S_1;\ S_2, \sigma\rangle \to \sigma'}$$

$$\text{(Assign)}\ \frac{\gamma, \sigma \vdash e \to v}{\gamma \vdash \langle x = e, \sigma\rangle \to \sigma[x \mapsto v]}$$

$$\text{(Let)}\ \frac{x'\ fresh \quad \gamma, \sigma \vdash e \to v \quad \gamma \vdash \langle [x'/x]S, \sigma[x' \mapsto v]\rangle \to \sigma'\ |_\sigma}{\gamma \vdash \langle \texttt{let}\ x : t = e\ \texttt{in}\ \{S\}, \sigma\rangle \to \sigma'}$$

Where $[x'/x]S$ is a capture-avoiding substitution (see Section 2.3.1), which substitutes all instances of x with the fresh variable x' in S, such that x' merely shadows x instead of directly replacing it. The idea behind this *let* syntax is to define variables alongside the new scope, such that

```
let x: int = 42 in {
    // x is local here
} // x no longer exists
```

This makes it easier to reason about with which scope a given variable is associated. For this example assume that $\sigma = \{output \mapsto (0, \texttt{int})\}$:

```
let x: int = 42 in {
    let y: int = 13 in {
        x = x + y;
    } // y is destroyed

    output = x;
} // x is destroyed
```

The shadowing mechanic simply permits the following behaviour:

```
let x: int = 42 in {
    let x: str = "hello" in {
        // x is "hello" here
    }
    // x is once again 42 here
}
```

This behaviour is desirable because it means any given variable only ever relates back to its nearest surrounding scope. Two xs can be defined simultaneously without having to syntactically disambiguate them. If the user wanted to reference them both within the same scope, it then stands to reason they should be called different things.

Sugar As a point of convenience, we also define two alternate syntaxes:

```
{
    // code goes here
}
```

To be a scope with no associated variable, and

```
let x: int = 42, y: str = "hello" in {
    // x and y are local here
} // x and y no longer exists
```

to be syntactic sugar for

```
let x: int = 42 in {
    let y: str = "hello" in {
        // x and y are local here
    }
} // x and y no longer exists
```

Both `x` and `y` are created and destroyed together. If one wants to add statements between the two scopes, they would have to be split up.

2.3.1 Capture-Avoiding Substitution

Formally, we define the capture avoiding substitution for statements as follows:

$$
\begin{aligned}
&[x'/x]\texttt{skip} = \texttt{skip} \\
&[x'/x](\texttt{let } x\texttt{: } t \texttt{ = } e \texttt{ in } \{S\}) = \texttt{let } x'\texttt{: } t \texttt{ = } [x'/x]e \texttt{ in } \{[x'/x]S\} \\
&[x'/x](S_1\texttt{; } S_2) = [x'/x]S_1\texttt{; } [x'/x]S_2 \\
&[x'/x](\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) = \texttt{if } [x'/x]e \texttt{ then } [x'/x]S_1 \texttt{ else } [x'/x]S_2 \\
&[x'/x](\texttt{while } e \texttt{ do } S) = \texttt{while } [x'/x]e \texttt{ do } [x'/x]S \\
&[x'/x](x \texttt{ = } e) = [x'/x]x \texttt{ = } [x'/x]e
\end{aligned}
$$

However, since variables can also occur in expressions, we need a similar set of rules for those:

$$
\begin{aligned}
&[x'/x]x = x' \\
&[x'/x](\Diamond_u\ e) = \Diamond_u\ [x'/x]e \\
&[x'/x](e_1\ \Diamond_b\ e_2) = [x'/x]e_1\ \Diamond_b\ [x'/x]e_2 \\
&[x'/x](e_1, e_2) = ([x'/x]e_1, [x'/x]e_2) \\
&[x'/x]first(e) = first([x'/x]e) \\
&[x'/x]second(e) = second([x'/x]e)
\end{aligned}
$$

2.4 Expression Semantics

$$\text{(Var Lookup)}\ \frac{}{\gamma, \sigma \vdash x \to v}\ \text{where } (v, t) = \sigma[x]$$

$$\text{(Binary Operator)}\ \frac{\gamma, \sigma \vdash e_1 \to v_1 \quad \gamma, \sigma \vdash e_2 \to v_2}{\gamma, \sigma \vdash e_1\ \Diamond_b\ e_2 \to v}\ \text{where } v = v_1\ \Diamond_b\ v_2$$

$$\text{(Unary Operator)}\ \frac{\gamma, \sigma \vdash e_1 \to v_1}{\gamma, \sigma \vdash \Diamond_u\ e_1 \to v}\ \text{where } v = \Diamond_u\ v_1$$

$$\text{(Tuples)}\ \frac{\gamma, \sigma \vdash e_1 \to v_1 \quad \gamma, \sigma \vdash e_2 \to v_2}{\gamma, \sigma \vdash (e_1, e_2) \to (v_1, v_2)}$$

$$\text{(Tuple First)}\ \frac{}{\gamma, \sigma \vdash first(e) \to v}\ \text{where } \pi_l(e) = v$$

$$\text{(Tuple Second)}\ \frac{}{\gamma, \sigma \vdash second(e) \to v}\ \text{where } \pi_r(e) = v$$

2.5 Type System

As briefly mentioned before, we will also be defining a static type system for While. The purpose of this type system is to be able to make static guarantees about the programs, and ensure our type-assumptions are upheld. Recall the definition of *Type* from Section 2.1 which lists the valid types available in While:

$$type \in Type ::= \texttt{int} \mid \texttt{str} \mid (type, type)$$

We also define a new environment

$$\gamma \in \Gamma = Var \to Type$$

to be our variable environment in our type system. While similar to Σ, Γ only keeps track of variable types, and not their values[VIS96; Hüt10].

2.5.1 Expressions

$$\text{(Int)}\ \frac{}{\gamma \vdash v : int}\ \text{if } v \in \mathbb{Z}$$

$$\text{(String)}\ \frac{}{\gamma \vdash v : str}\ \text{if } v \in String$$

$$\text{(Tuple)}\ \frac{\gamma \vdash e_1 : t_1 \quad \gamma \vdash e_2 : t_2}{\gamma \vdash (e_1, e_2) : (t_1, t_2)}$$

$$\text{(Var)}\ \frac{}{\gamma \vdash x : t}\ \text{if } \gamma(x) = t$$

$$\text{(Binary operation)}\ \frac{\gamma \vdash e_1 : t_1 \quad \gamma \vdash e_2 : t_2}{\gamma \vdash e_1 \Diamond_b e_2 : t_3}$$

$$\text{(Unary operation)}\ \frac{\gamma \vdash e : t_1}{\gamma \vdash \Diamond_u e : t_2}$$

2.5.2 Statements

$$\text{(Reassign)}\ \frac{\gamma \vdash x : t \quad \gamma \vdash e : t}{\gamma \vdash x = e}$$

$$\text{(Concat)}\ \frac{\gamma \vdash S_1 \quad \gamma \vdash S_2}{\gamma \vdash S_1; S_2}$$

$$\text{(If)}\ \frac{\gamma \vdash e : Int \quad \gamma \vdash S_1 \quad \gamma \vdash S_2}{\gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}$$

$$\text{(While)}\ \frac{\gamma \vdash e : Int \quad \gamma \vdash S}{\gamma \vdash \texttt{while}\ e\ \texttt{do}\ S}$$

$$\text{(Skip)}\ \frac{}{\gamma \vdash \texttt{skip}}$$

$$\text{(Let)}\ \frac{\gamma \vdash e : t \quad \gamma[x \mapsto t] \vdash S}{\gamma \vdash \texttt{let}\ x : t = e\ \texttt{in}\ \{S\}}$$

2.6 Let in Depth

The way we structure and handle `let`-bindings may seem a bit strange coming from a more traditional programming language—Rust in particular. The reason is quite simple: it greatly simplifies the semantics for variable scoping by completely eliminating the need to have a scope stack.

In Rust it is possible to write

```
let x: i32 = 12;
let y: i32 = 15;

{ // introduce new local scope
    let y: i32 = 8; // introduce new local y
    let x: String = format!("{}", x + y); // introduce new local x

    println!("{}", x); // prints "20"
} // local x and y destroyed here

println!("{}", x); // prints 12
```

While this example is a little contrived, it should be clear that the way scopes are handled in Rust can make keeping track of variables a bit hairy at best. To remedy this, our `let`-bindings instead attach a scope to the relevant variable explicitly. This does two things: 1) it eliminates the need of a scope stack, any bound variable only ever exists in its own local scope, and 2) it clears up the possible ambiguity there might be when dealing with sequentially defined variables and their inter-variable scope dependencies.

The following is how the very same example would be written in our toy language:

```
1 let x: int = 12, y: int = 15 {
2     let y: int = 8, x: str = str(x + y) { // local inner scope with new x and y
3         print(x); // x is printed, prints "20"
4     }
5     print(x); // x is printed, prints 12
6 }
```

Note that this example assumes the existence of a `print` and `str` function, which print to standard output and converts to a string respectively.

We have derived the type tree and the semantic tree from the example in Figure 1 and Figure 2 in Appendix A

And the σ will develop as so throughout the program:

$$1 : \sigma[x' \mapsto 12, y' \mapsto 15] \tag{1}$$
$$2 : \sigma[y'' \mapsto 8, x'' \mapsto 20] \tag{2}$$

Because of capture avoiding substitution it becomes x', y' and x'', y'' and not just x, y. This is done to allow shadowing of variables and to not overwrite the variables values once you get out of the shadowing scope.

2.7 Security

Our While language is a simple language which has some of the same structures and flows of real programming languages. In the While language it is easily to accidentally leak the secure information.

```
let x:str = "secret" in {
    let used:str = x in {
        let leak:str = x {
            //do some other stuff
        }
    }
}
```

In the example we could easily use the secret information in x and after x has been used, leak the information. We could also have leak the secret information in x first and then used x, the semantic is not stopping us from that.

Lets us try to make the example from Section 1 where we handle a users information.

```
let user_info: str = input in {
    user_info = sanitize(user_info);
    let result: str = "" in {
        let access_key: str = crypto(user_info) in {
            let access_token: str = lookup(access_key) in {
                if access_token == "ok" then
                    result = "access granted";
                else
                    result = "access denied";
            }
        }
        output = result;
    }
}
```

We have here written the example in our while language. To most people, the above example looks fine, but nothing in this prevents a malicious user from leaking any of the user info at any stage of the program. Assuming the existence of a variable named *debug*, it is not unreasonable to imagine the following scenario:

```
// code above unchanged
let access_key: int = crypto(user_info) in {
    debug = access_key;
    let access_token: str = lookup(access_key) in {
        // code below unchanged
```

Which is then left unremoved by mistake when the software is moved to production.

One means of combatting these sorts of oversights is the concept of *ownership* as implemented by the Rust Programming Lanugage. This can make it more difficult to leak information unintended, and makes some other guaranties. We will like to model Rusts ownership model in our While language to see if it can have some improvement on information security in a programming language.

3 Ownership

As discussed in [Wei+19], one of Rust's most fundamental features is its way of dealing with the concept of *ownership* and the implications this has on how code is written. Put simply, the purpose of ownership is to statically keep track of variables and references throughout the scopes of a given program. The principle builds on giving a variable full ownership over a given value, and once that variable leaves the scope in which it is declared, the value and corresponding variable gets destroyed. Ownership to a value can also be transferred to new variables; doing so will invalidate the old variable. The purpose of this is to ensure a clear single source of truth so that values can easily be traced back to the source. Having the old variable get invalidated every time it gets reassigned does cause problems sometimes. To this end, ownership permits a form of *borrowing* of variables, where references to a given variable are bound to a new variable. Once the new variable leaves a scope, the reference is automatically freed. However, allowing for multiple simultaneous mutable references violates the principle of a "clear single source of truth", which is why the ownership model only allows a single mutable reference to be present at any given time[Jun+18].

We will now extend our while language to model ownership as it is found in Rust.

3.1 Syntax & Set Extensions

We will extend the syntax of our while language as below:

$$
\begin{aligned}
stmt \in Stmt &::= \cdots \\
&\mid *x = expr && \text{(Deref Assignment)} \\
expr \in Expr &::= \cdots \\
&\mid \&x && \text{(Shared Variable)} \\
&\mid @x && \text{(Unique Variable)} \\
type \in Type &::= \cdots \\
&\mid @type && \text{(Mutable Unique References)} \\
&\mid \&type && \text{(Immutable Shared References)}
\end{aligned}
$$

To this we also add the following unary operators: $*, \&, @$ for dereferencing, shared referencing, and uniquely referencing respecitvely.

& and @ Rust famously has two different kinds of references: immutable reference—called *borrows*—which are written `&T`, and mutable references, written `&mut T` [KN18]. In C++ parlance, these are referred to as *shared* and *unique* references respecitvely; this naming scheme better reflects the *purpose* of the references, and thus is what we choose to call them.

& is the *immutable shared reference*. Many of these references can point to the same source variable, aliasing it immutably. As long as the data remains immutable, the single source of truth principle is not violated. One can think of this as giving multiple processes read access to the same file: as long as no process changes the file, overall consensus on the file's contents is guaranteed.

@ is the *mutable unique reference*. As its namesake suggests, this kind of reference must always be unique; if more than one exists, both references could potentially mutate the source variable, potentially causing unexpected results. To go back to the file analogy: while giving multiple processes read access is fine, once as much as one of the processes needs to write to the file, the single source of truth principle could be violated unless it is the only process with access.

Values We redefine our values to include Var so that we can have a reference to a variable.

$$Val = \mathbb{Z} \cup String \cup Var \cup (Val \times Val)$$

Capture-Avoiding Substitution We add this to the Capture-Avoiding Substitution rules.

$$
\begin{aligned}
[x'/x]@x &= @x' \\
[x'/x]\&x &= \&x' \\
[x'/x] * x &= *x'
\end{aligned}
$$

Copy We also define a subset of $Type$, called $Copy$

$$copy \in Copy ::= \texttt{int} \mid (copy, copy)$$

Which is the set of *copyable* types that do not obey the standard move semantics. The motivation for these, is that—as we will explain more in depth later—without *copy types*, doing simple arithmetics with variables would become incredibly cumbersome. Having integers simply be copyable would make expressions like $x = x + x - 3$ doable without having to use references. Note that since `int` is the only simple type in $Copy$, it means that $(\texttt{int}, \texttt{str}) \notin Copy$ while $(\texttt{int}, (\texttt{int}, \texttt{int})) \in Copy$.

In Rust, implementing $Copy$ for a type tells the compiler that it is safe to perform shallow bit-wise copies on the types that implement it. Most primitive and simple types, as well as arrays and tuples containing primitive types implement $Copy$[KN18]. It is generally safe to perform bit-wise copies on primitive data, since no references end up duplicated. However, this also means that references by definition cannot implement $Copy$.

Vars We also define a function $vars$, which operates on the $Expr$ set defined in Section 2.1

$$\begin{aligned}
&vars : Expr \to \mathcal{P}(Var) \\
&vars(v) = \emptyset && \text{if } v \text{ is a value} \\
&vars(x) = \{x\} && \text{if } x \text{ is a variable} \\
&vars(e_1 \diamondsuit_b e_2) = vars(e_1) \cup vars(e_2) \\
&vars(\diamondsuit_u e) = vars(e) \\
&vars((e_1, e_2)) = vars(e_1) \cup vars(e_2) \\
&vars(*y) = \{y, x\} && \text{where } \sigma(y) = \&x \\
&vars(*y) = \{y, x\} && \text{where } \sigma(y) = @x
\end{aligned}$$

which *pulls out* all the variables used within an expression.
For example, $vars($`x + y - 3 * (z + x)`$) = \{x, y, z\}$.

The purpose of this function is to aid in cleaning up the environment of variables whose values have been moved.

```
let hello: str = "Hello " in {
    let world: str = "world" in {
        hello = hello ++ world; // world variable removed here due to move.
    }
}
```

Reference Counting We also need three functions to count the references of a given variable in the environment:

$$\begin{aligned}
&rc_{\&} : Var \times \Sigma \to \mathbb{N} \\
&rc_{\&}(y, \sigma) = |\{x \mid x \in Var \wedge \&y = \sigma(x)\}|
\end{aligned}$$

$$\begin{aligned}
&rc_{@} : Var \times \Sigma \to \mathbb{N} \\
&rc_{@}(y, \sigma) = |\{x \mid x \in Var \wedge @y = \sigma(x)\}|
\end{aligned}$$

$$\begin{aligned}
&rc_{all} : Var \times \Sigma \to \mathbb{N} \\
&rc_{all}(y, \sigma) = rc_{\&}(y, \sigma) + rc_{@}(y, \sigma)
\end{aligned}$$

$rc_{\&}$ counts all the & references of an given variable in a given environment and so does $rc_{@}$ for @ references. We denote $\&t$ and $@t$ as a type name prefixed with the given reference operator. The general function rc_{all} counts all references regardless of kind.

Counting references is important for the upcoming semantics, as we need to make sure certain reference-related invariants are upheld. Adding an invariant that limits the number of shared references simply becomes a matter of writing $rc_{\&}(x, \sigma) \leq 3$ as one of the hypotheses.

3.2 Extended Semantics

We will extend and redefine some of the semantics of our while language.

$$\text{(Assign-Copy)}\ \frac{\gamma, \sigma \vdash e \to v \quad rc_{all}(x, \sigma) = 0 \quad \gamma \vdash x : t \quad t \in Copy}{\gamma \vdash \langle x = e, \sigma \rangle \to \sigma[x \mapsto v]}$$

$$\text{(Assign-Move)}\ \frac{\gamma, \sigma \vdash e \to v \quad rc_{all}(x, \sigma) = 0 \quad \gamma \vdash x : t \quad t \notin Copy}{\gamma \vdash \langle x = e, \sigma \rangle \to \sigma'[x \mapsto v]}\ \text{where } \sigma' = \sigma \setminus vars(e)$$

Two new rules for assignment have been created which will affect every rule that in some way uses values and variables. The two rules Assign-Move and Assign-Copy differ in the way they handle the *Copy* subtype. A variable which type is in Copy can be used by multiple times without its ownership is getting transfered, that is to say, the value is *copied* to its new owner, rather than moved. In contrast, variables not in *Copy* cannot be used by the original variable once before the ownership is transfered.

We also need to be able to follow a reference from one variable to another, this is done simply by looking the variable up as normal, then stripping the reference symbol:

$$\text{(Ref Lookup \&)}\ \frac{\sigma(x) = y \quad \gamma \vdash x : \&t}{\gamma, \sigma \vdash x \to_r y}$$

$$\text{(Ref Lookup @)}\ \frac{\sigma(x) = y \quad \gamma \vdash x : @t}{\gamma, \sigma \vdash x \to_r y}$$

$$\text{(Let Copy)}\ \frac{\begin{array}{c} x'\ fresh \quad \gamma, \sigma \vdash e \to v \\ \gamma \vdash \langle [x'/x]S, \sigma[x' \mapsto v] \rangle \to \sigma' \mid_{\sigma} \quad t \in Copy \quad rc_{all}(x', \sigma') = 0 \end{array}}{\gamma \vdash \langle \texttt{let}\ x : t = e\ \texttt{in}\ \{S\}, \sigma \rangle \to \sigma'}$$

$$\text{(Let Move)}\ \frac{\begin{array}{c} x'\ fresh \quad \gamma, \sigma \vdash e \to v \\ \gamma \vdash \langle [x'/x]S, \sigma''[x' \mapsto v] \rangle \to \sigma' \mid_{\sigma} \quad t \notin Copy \quad rc_{all}(x', \sigma') = 0 \end{array}}{\gamma \vdash \langle \texttt{let}\ x : t = e\ \texttt{in}\ \{S\}, \sigma \rangle \to \sigma'}\ \text{where } \sigma'' = \sigma \setminus vars(e)$$

$$\text{(Deref-Assign Cpy)}\ \frac{\begin{array}{c} \gamma, \sigma \vdash x \to_r y \quad \gamma, \sigma \vdash e \to v \\ rc_{@}(y, \sigma) = 1 \quad rc_{\&}(y, \sigma) = 0 \quad \gamma \vdash x : t \quad t \in Copy \end{array}}{\gamma \vdash \langle *x = e, \sigma \rangle \to \sigma[y \mapsto v]}$$

$$\text{(Deref-Assign Mov)}\ \frac{\begin{array}{c} \gamma, \sigma \vdash x \to_r y \quad \gamma, \sigma \vdash e \to v \\ rc_{@}(y, \sigma) = 1 \quad rc_{\&}(y, \sigma) = 0 \quad \gamma \vdash x : t \quad t \notin Copy \end{array}}{\gamma \vdash \langle *x = e, \sigma \rangle \to \sigma'[y \mapsto v]}\ \text{where } \sigma' = \sigma \setminus vars(e)$$

Deref Assign is a rule for when you have stored a reference to variable y in x and wants to write a value to y you can write $*x = e$ to overwrite the value of y with whatever is in e. Notice how Deref-Assign Mov deletes the moved variables after assignment, while Deref-Assign Cpy does not.

```
let y: int = 42 {
    let x: @int = @y {
        *x = 43; // y -> 43
    }
}
```

In this example, we define y to be 42 and x to be an unique reference to y. Then we assign 43 to the derefed x witch then makes $y = 43$.

$$\text{(Shared Ref)}\ \frac{rc_{@}(y, \sigma) = 0}{\gamma, \sigma \vdash \&y \to v}\ \text{where } v \text{ is a ref of } \&y$$

Note the invariant $rc_{@}(y, \sigma) = 0$ which specifies that no unique references must exist. This falls in-line with the *single-source-of-truth* principle. As long as there is no writer, a potentially infinite number of readers can be made.

```
let y: int = 42 {
    let x: &int = &y {

    }
}
```

In this example, we define y to be 42, and x to be a shared reference to y. Note that shared references do not implement any variant of Deref-Assign, as such their values can only be read.

$$\text{(Unique Ref)}\ \frac{rc_{all}(y, \sigma) = 0}{\gamma, \sigma \vdash @y \to v}\ \text{where } v \text{ is a ref of } @y$$

The invariant $rc_{all}(y, \sigma) = 0$ simply specifies that *no* prior references of any kind must exist for a unique reference to be created. This is done to ensure the uniqueness of the reference.

```
let x: int = 15 {
    let y: @int = @x {
        let z: @int = @x { // ERROR: reference no longer unique.

        }
    }
}
```

In this example, we define x to be 15 and y to be a unique reference to x. Then we try to define z to be another unique reference to x, but that is not allowed because there already is an unique reference to x.

$$\text{(Deref)}\ \frac{\gamma, \sigma \vdash y \to_r y' \qquad \gamma, \sigma \vdash y' \to v}{\gamma, \sigma \vdash *y \to v}$$

```
let x: int = 2 {
    let y: &int = &42 {
        x = x + *y; // x -> 44
    }
}
```

In this example, we define x to be 2 and y to be a shared reference to 42. Then we assign x to be $x + *y$ the derefered y.

```
let x: int = 7 {
    let y: &int = &x {
        let z: @int = @x { //ERROR: there is already a shared reference

        }
    }
}
```

In this example, we define x to be 7, and then y to be a shared reference to x, if we then try to define z to be a unique reference to x we will get an error, because we already have a shared reference to x. This rule applies the other way around too, so that if there is a unique reference to x then an error will occur if more shared references are made afterwards.

3.3 Type System

We will here extend our typesystem for our while language.

3.3.1 Expressions

$$(@\textsc{Var})\ \frac{}{\gamma \vdash x : @t}\ \text{if } \gamma(x) = @t$$

$$(\&\textsc{Var})\ \frac{}{\gamma \vdash x : \&t}\ \text{if } \gamma(x) = \&t$$

3.3.2 Statements

$$(\textsc{Assign})\ \frac{\gamma \vdash x : t \quad \gamma \vdash e : t}{\gamma \vdash x = e}$$

$$(\textsc{Deref-Assign})\ \frac{\gamma \vdash e : t \quad \gamma \vdash x : @t}{\gamma \vdash *x = e}$$

3.4 Security

In Section 2.7 we have shown how easily we unintended can leak secret information without our semantics or type system giving us any warning. We have now defined the ownership semantics and type system, and put it into the While language.

```
let x: str = "secret" in {
    let used: str = x in {
        let leak: str = x in { //error x has been moved

        }
    }
}
```

If we try to simply leak the secret information as in Section 2.7 we will get an error because x has been moved to *used*, and if we try to assign x to *leak* before *used* we then have the same problem, x has been moved. We can no longer unintended try to leak our secret information because when we first have moved the secret information it can no longer be used again in another context.

But Rust has implemented a function called *clone* on most built-in types witch takes a variable and returns its value without taking over its ownership. Clone acts as a *deep copy* counterpart to *Copy*, as it also takes care to resolve and duplicate any internal references[KN18]. If we assume that such a function exists and we would be able to do so.

```
let x: str = "secret" in {
    let used: str = clone(x) in {
        let leak: str = clone(x) in {

        }
    }
}
```

It is now legal for us to *clone* x into *used* and then again clone x to *leak* and this way *circumvent* the ownership model. We would then have to use a taint analysis to find runs that that leaks the secret information.

A point in favour of *clone* however, is that it makes these sorts of actions explicit for the programmer, thus it is impossible to break ownership without being explicit about it.

When we look at the example from Section 1 and 2.7 we see that there is no need to change anything for it to work. But we get the security that the variables cannot be moved without an error occur, so if we accidently try to leak a variable we are not supposed to, we get an error.

```
let user_info: str = input in {
    user_info = sanitize(user_info);
    let result: str = "" in {
        let access_key: str = crypto(user_info) in {
            let access_token: str = lookup(access_key) in {
                if access_token == "ok" then
                    result = "access granted";
                else
                    result = "access denied";
            }
        }
        output = result;
    }
}
```

That is—of course—unless the value is explicitly cloned.

To remedy this, we can try and wrap the context in an opaque data type that—even if cloned—would be useless to an attacker.

4 SecWhile

The paper *A Library for Light-Weight Information-Flow Security in Haskell* posits the creation of a *Sec* monad, which aims to encapsulate data within a monad with an associated security level. This security level ensures data cannot be leaked and can only ever be modified within the context of the monad [Wad92; RCH08]. However, there is one aspect we explicitly wish to not model from this library: the immutability and side-effect free nature of Haskell. Since Rust is not side-effect free, it would be antithetical for us to try and model SecWhile to be more like Haskell.

To get a better understanding of the security benefits of the ownership model, we wish to implement this notion of security into the language. To do this, we add the security levels *low* and *high* to the language:

$$\begin{aligned} \mathit{level} \in \mathit{Level} &::= \texttt{high} \mid \texttt{low} \\ \mathit{type} \in \mathit{Type} &::= \cdots \mid \texttt{sec(}\mathit{level}\texttt{, }\mathit{type}\texttt{)} \end{aligned}$$

We also specify that $\forall l \in \mathit{Level}.\ \texttt{low} \sqsubseteq \texttt{high}$. For the sake of simplicity, we choose to just stick to two security levels, but this could in theory be extended into a more complex lattice.

We will develop a security monad into the while language.

We also add the following constructs to *expr*

$$\begin{aligned} \mathit{expr} \in \mathit{Expr} ::= {} & \cdots \\ & \mid \texttt{mksec(}\mathit{expr}\texttt{, }\mathit{level}\texttt{)} && \text{(Constructor)} \\ & \mid \texttt{open(}\mathit{expr}\texttt{, }\mathit{level}\texttt{)} && \text{(Open)} \\ & \mid \texttt{map(}x\texttt{: }\mathit{type}\texttt{ -> }\mathit{expr}\texttt{, }\mathit{expr}\texttt{)} && \text{(Map)} \\ & \mid \texttt{flatten(}\mathit{expr}\texttt{)} && \text{(Flatten)} \end{aligned}$$

`mksec` MkSec acts as the monadic constructor for security levels, such that the same constructor can be used to make both levels. For example, `let x: sec(high, int) = mksec(42, high) {...}` is a valid assignment with a security level with $\{x \mapsto \mathit{sec}(42)\}$ as the environment.

`open` Open exists to be able to *unwrap* a value and bring it back out of the context of a security level. Ideally, it should only be used in a context where the user knows for certain that it is safe to bring the value out of a security level.

`map` Map lets us modify the value within a security level without also changing the level, for example let $\sigma = \{x \mapsto (42, \texttt{sec(high, int)})\}$, then `x = map(y: int -> y + 2, x)` would change it to $\{x \mapsto (44, \texttt{sec(high, int)})\}$.

`flatten` Flatten *squashes* nested security levels by one level of nesting up to their heigst common denominator. For example, given a nesting of `sec(high, sec(low, int))`, flatten would produce `sec(high, int)`, but a nesting of `sec(low, sec(low, int))` would produce a `sec(low, int)`.

4.0.1 Capture-Avoiding Substitution

We add this to the Capture-Avoiding Substitution rules.

$$
\begin{aligned}
&[x'/x]mksec(e) = mksec([x'/x]e) \\
&[x'/x]open(e, l) = open([x'/x]e, l) \\
&[x'/x]map(x : t- > e, e) = map(x' : t- > [x'/x]e, [x'/x]e) \\
&[x'/x]flatten(e) = flatten([x'/x]e)
\end{aligned}
$$

4.1 Semantics

The semantics for the new syntax is relatively straightforward, the only thing to really note here, is the use of `mksec` in the context of a `let` statement. It will construct whichever security level specified in the type of `let`.

$$\text{(MkSec)}\ \frac{\gamma, \sigma \vdash e \to v}{\gamma, \sigma \vdash \texttt{mksec(}e\texttt{, } l\texttt{)} \to \texttt{sec(}v\texttt{)}}$$

$$\text{(Open)}\ \frac{\gamma, \sigma \vdash e \to \texttt{sec(}v\texttt{)}}{\gamma, \sigma \vdash \texttt{open(}e\texttt{, } l\texttt{)} \to v}$$

$$\text{(Map)}\ \frac{\gamma, \sigma \vdash e_1 \to \texttt{sec(}v\texttt{)} \quad \gamma, \sigma[x \mapsto v] \vdash e_2 \to v'}{\gamma, \sigma \vdash \texttt{map (}x\texttt{: } t\texttt{ -> } e_2\texttt{, } e_1\texttt{)} \to \texttt{sec(}v'\texttt{)}}$$

$$\text{(Flatten)}\ \frac{\gamma, \sigma \vdash e \to \texttt{sec(sec(}v\texttt{))}}{\gamma, \sigma \vdash \texttt{flatten(}e\texttt{)} \to \texttt{sec(}v\texttt{)}}$$

Note that whenever we write `sec(`e`)` or `sec(`v`)` we mean an expression or value wrapped in a security level, rather than it being a syntactic construct.

4.2 Type System

Here we extend our type system

$$\text{(MkSec)}\ \frac{\gamma \vdash e : t \quad l \in Level}{\gamma \vdash \texttt{mksec(}e\texttt{, } l\texttt{):sec(}l\texttt{, } t\texttt{)}}$$

$$\text{(Map)}\ \frac{\gamma \vdash e_1 : sec(l, t_1) \quad \gamma \vdash e_2 : t_2 \quad l \in Level}{\gamma \vdash \texttt{map(}e_1\texttt{, } x\texttt{: } t_1\texttt{ -> } e_2\texttt{)} : \texttt{sec(}l\texttt{, } t_2\texttt{)}}$$

$$\text{(Open)}\ \frac{l \sqsubseteq l' \quad l, l' \in Level \quad \gamma \vdash e : sec(l, t)}{\gamma \vdash \texttt{open(}e\texttt{, } l'\texttt{)} : t}$$

$$\text{(Flatten)}\ \frac{\gamma \vdash e : \texttt{sec(}l_1\texttt{, sec(}l_2\texttt{, } t\texttt{))} \quad l_1, l_2 \in Level}{\gamma \vdash \texttt{flatten(}e\texttt{)} : \texttt{sec(}l\texttt{, } t\texttt{)}} \text{ where } l = l_1 \sqcup l_2$$

4.3 DoSec

We will introduce a secure environment where it can safely open a security monad and execute command on its value, without taking greater risk at exposing the value in the secure monad.

We call this environment `do_sec` witch takes a security level and a statement, witch will be executed under the security level.

4.3.1 Semantics

$$(\textsc{DoSec})\ \frac{\gamma \vdash \langle S, \sigma \rangle}{\gamma \vdash \langle \texttt{do\_sec(}l\texttt{)}\ \ S, \sigma \rangle}$$

4.3.2 Type System

We can make all the necessary changes for *do_sec* to work in the type system. We make an environment constant called λ which will save the security level. At the start of the program λ will be *low* and can be set to *high* within do_sec.

Statements This will changes some conditions and make the do_sec secure.

$$(\textsc{DoSec})\ \frac{\gamma[\lambda \mapsto l] \vdash S}{\gamma \vdash \texttt{do\_sec(l) S}}$$

We will redefine the assign and let rules and add type rules for the security monad.

$$(\textsc{AssignSec})\ \frac{\gamma(\lambda) = l \quad \gamma \vdash e : t \quad \gamma \vdash x : sec(l,t)}{\gamma \vdash \texttt{x = e}}\ \text{if } l \sqsupset low$$

This ensures that we only can assign to a security monad inside a do_sec.

$$(\textsc{AssignUnSec})\ \frac{\gamma(\lambda) = l \quad \gamma \vdash e : t \quad \gamma \vdash x : t}{\gamma \vdash \texttt{x = e}}\ \text{if } l = low$$

And outside a do_sec everything should be handled as normal.

$$(\textsc{LetSec})\ \frac{\gamma(\lambda) = l \quad \gamma \vdash e : t \quad \gamma[x \mapsto sec(l,t)] \vdash S}{\gamma \vdash \texttt{let x: t = e in S}}\ \text{if } l \sqsupset low$$

This makes all let definition inside a do_sec to be a security monad.

$$(\textsc{LetUnSec})\ \frac{\gamma(\lambda) = l \quad \gamma \vdash e : t \quad \gamma[x \mapsto t] \vdash S}{\gamma \vdash \texttt{let x: t = e in S}}\ \text{if } l = low$$

And outside the do_sec everything should be handled as normal.

Expressions

$$(\textsc{Sec})\ \frac{}{\gamma \vdash \texttt{x} : sec(l,t)}\ \text{if } \gamma(x) = sec(l,t)$$

$$(\textsc{MkSec})\ \frac{\gamma \vdash e : t}{\gamma \vdash \texttt{mksec(}e\texttt{): t}}$$

4.3.3 Examples

```
1 let y:str = "" in {
2     let x:sec(high,str) = mksec("secret") in {
3         do_sec(high){
4             y = x; //type error
5         }
6     }
7 }
```

We get a type error because of the rule AssignSec, the λ is set to high, to every assignment may only be made to variables of type sec(high, t). And y is only of type str. This also prevent indirect leaking, when x is used in an if-statement, you still cannot assign to y for the same reason as before.

To see why, let us step through the program's memory a line at a time:

1. A new variable y is defined, which sets $\gamma = \{y \mapsto str\}$.

2. The variable x is defined, $\gamma = \{y \mapsto str, x \mapsto sec(high, str)\}$.
3. The program enters the Sec monad, $\gamma = \{y \mapsto str, x \mapsto sec(high, str), \lambda \mapsto high\}$.
4. The program assigns x to y, which is in violation of AssignSec due to y not being of type $sec(high, str)$.
5. Exit Sec monad.
6. Exit *let* x.
7. Exit *let* y.

4.3.4 Capture-Avoiding Substitution

We add this to the Capture-Avoiding Substitution rules.

$$[x'/x](\texttt{do\_sec}(l)\ \ S) = \texttt{do\_sec}(l)\ \ [x'/x]S$$

4.4 Security

In section 3.4 we have shown how ownership can prevent unintended leaking of secure information, but also how we could overcome the ownership model by Rust's *clone* function.

By implementing the security model from [RCH08] into the while language, we can prevent secure information from being leaked as long as the secure information is inside a security monad.

```
let x:sec(high,str) = mksec("secret") in {
    let leak:str = "" in {
        do_sec(high){
            let used:str = x in {
                leak = x; //type error
            }
        }
    }
}
```

When we are inside a *do_sec* every assignment must be of type $sec(l, t)$ where l is the security level of the *do_sec*, but *leak* is only of type *str*, thus incompatible.

If we would want to open the security monad, but this is done explicitly and can only be done if clearly intended.

```
let x:sec(high,str) = mksec("secret") in {
    let leak:str = open(x, high) in {

    }
}
```

If we look at the example from Section 1, 2.7, and 3.4, and try to write it with the security monad and the *do_sec* we get an more intended way of controlling the security of the program, so we do not untended leak secure information.

```
let user_info: str = input in {
    user_info = sanitise(user_info);
    let sec_info: sec(high, str) = mksec(user_info) in {
        let result: sec(high, str) = mksec") in {
            do_sec(high) {
                let access_key: str = crypto(sec_info) in {
                    let access_token: str = lookup(key) in {
                        if token == "ok" then
                            result = "access granted"
                        else
                            result = "access denied";
```

```
                    }
                }
            }
            output = open(result, high);
        }
    }
}
```

Even if we were to clone the access key, the data would not be able to escape the security monad, and thus effectively be useless. As in [RCH08], we assume only the programmer with the correct permissions has access to the *open* function, such that unintended access is minimised.

5 Discussion

We have made a While language to simplify some of the features in a modern programming language. In simple programming languages that facilitate side-effects—like our While language—It is easy to make security-related mistakes and leak data by accident. To remedy this we have attempted to combine a security monad with the ownership model to see if decent security can be achieved in a language with side-effects.

While In our While language it it easy to leak a secure information unintended and implicit. So a programmer without much experience can easily make some security mistakes and unintended leak the secure information the program is handling.

Ownership We then added ownership from Rust to our While language so that a variables ownership can be moved and borrowed. This makes it more difficult to accidently leak a variable, because when a variable's ownership is moved when leaked, the variable cannot be used later on unless explicitly cloned. This inclusion of explicitness alerts the programmer to the possibility that they might be doing something bad, thus adding a first line of defence against bone-headed mistakes.

SecLib We have attempted to add an implementation of SecLib from our last project [RL19] to our While language, such that we can put secret information into a security monad. One can only open a security monad by using its own security level, this way you can not unintended open the monad and leak the secure information, you will have to do it specific. Additionally we have added a *do_sec* where you can alter the secure information but you can not leak it or in any way get it out of the security monad without doing it explicit and intended.

6 Conclusion

Working in a language without side-effect-free programming, there are certain guarantees, which cannot be met. In Haskell, everything is laid out in the type: if I/O is present, the type reflects it; if stateful operations are present, the type also reflects that. This creates the guarantee that if something were to ever change the state of the world, it would be reflected in the types. This, in combination with a security monad[RCH08] could then help limit unintended data leakage. Rust does not have any of that. What Rust does have, is ownership, a property which lets the programmer write software that includes side-effects, but in a somewhat controlled and constrained manner. Our goal with this report was to find out if ownership could stand in for explicit side-effects and help making program flow more secure and it turns out that it can, but only if the programmer in question chooses not to explicitly clone and leak the sensitive data.

We also found, that some of the security guarantees are less than ideally fulfilled in Rust without the existence of a monad mechanism. While monads could likely be simulated via a combination of macros and methods, this can likely not stand in for the true expressivity of a proper context block like our `do_sec` (see Section 4.3).

References

[Hüt10] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN: 978-0-521-14709-5. URL: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/transitions-and-trees-introduction-structural-operational-semantics.

[Jun+18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the rust programming language". In: *PACMPL* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. first. No Starch Press, 2018. ISBN: 987-1-59327-828-1.

[RL19] Morten Rasmussen and Nikolaj Lepka. "Static Information Flow Security in Rust". In: (2019).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2/3 (1996), pp. 167–188. DOI: 10.3233/JCS-1996-42-304. URL: https://doi.org/10.3233/JCS-1996-42-304.

[Wad92] Philip Wadler. "Comprehending Monads". In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 461–493. DOI: 10.1017/S0960129500001560. URL: https://doi.org/10.1017/S0960129500001560.

[Wei+19] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. "Oxide: The Essence of Rust". In: *CoRR* abs/1903.00982 (2019). arXiv: 1903.00982. URL: http://arxiv.org/abs/1903.00982.

A Prooftree

$$\dfrac{\dfrac{}{\gamma \vdash 12 : int}\ \text{(Int) if } 12 \in \mathbb{Z} \qquad \text{(Let)}\ \dfrac{\dfrac{}{\gamma \vdash 15 : int}\ \text{(Int) if } 15 \in \mathbb{Z} \qquad \text{(Let)}\ \dfrac{\dfrac{}{\gamma \vdash 8 : int}\ \text{(Int) if } 8 \in \mathbb{Z} \qquad \text{(Let)}\ \dfrac{\dfrac{\dfrac{}{\gamma \vdash x : int}\ \text{(Var) if } \gamma(x) = t \qquad \dfrac{}{\gamma \vdash y : int}\ \text{(Var) if } \gamma(y) = t}{\gamma \vdash x + y : int}\ \text{(Binary Operation)}}{\gamma[y \mapsto int] \vdash \texttt{let } y\texttt{:int = } 8 \texttt{ in } S} }{\gamma[x \mapsto int] \vdash \texttt{let } y\texttt{:int = } 15 \texttt{ in } S}}{\gamma \vdash \texttt{let } x\texttt{:int = } 12 \texttt{ in } S}$$

Figure 1: The type tree of the example in Section 2.6

$$\dfrac{x' fresh \quad \gamma \vdash 12 \to 12 \qquad \dfrac{y' fresh \quad \gamma \vdash 15 \to 15 \qquad \dfrac{y'' fresh \quad \gamma \vdash 8 \to 8 \qquad \dfrac{x'' fresh \qquad \dfrac{}{\gamma, \sigma \vdash x' \to 12}\ \text{(Var Lookup) where } 12 = \sigma[x'] \qquad \dfrac{}{\gamma, \sigma \vdash y'' \to 8}\ \text{(Var Lookup) where } 8 = \sigma[y'']}{\gamma, \sigma \vdash x' + y'' \to 20}\ \text{(Binary Operation)}}{\gamma \vdash [y''/y']\langle \texttt{let } x''\texttt{:int = } x' + y'' \texttt{ in S, } \sigma''[y'' \mapsto 8]\rangle \to \sigma''' \mid_{\sigma''}}\ \text{(Let)}}{\gamma \vdash [y'/y]\langle \texttt{let } y'\texttt{:int = } 8 \texttt{ in S, } \sigma'[y' \mapsto 15]\rangle \to \sigma'' \mid_{\sigma'}}\ \text{(Let)}}{\gamma \vdash [x'/x]\langle \texttt{let } y\texttt{:int = } 15 \texttt{ in S, } \sigma[x' \mapsto 12]\rangle \to \sigma' \mid_{\sigma}}\ \text{(Let)}}{\gamma \vdash \langle \texttt{let } x\texttt{:int = } 12 \texttt{ in S, } \sigma\rangle \to \sigma'}$$

Figure 2: The semantic tree of the example in Section 2.6