## Contents

# 1 Information Flow

## 1.1 Non-Interference

### 1.1.1 Motivation

Lets consider a simple program

```
x   = SECRET   # some secret data
y   = x
OBS = y        # observable data (e.g. by attackers)
```

Our essential goal is to prevent information about SECRET from flowing to OBS. However, in this example, information about SECRET flows to OBS through the variable y, i.e. there is a *direct flow*. Hence we would consider this program insecure or unsafe.
Lets look at another example

```
x = SECRET      # some secret data
OBS = 0         # public data
if x == 1:      # control flow depends on secret
    OBS = 1     # observable data (e.g. by attackers)
```

Now again we can ask ourselves here, is there a direct flow from SECRET to OBS? The answer is no, since y is not assigned any value depending on x. However, there is still an information flow from SECRET to OBS through the control flow of the program. This is called an *indirect flow* or *control flow leak*. Hence we would also consider this program unsafe.

### 1.1.2 The general idea

The commonality in the above examples is that we define a program to be *secure* if secret date does not 'interfere' with public or observable data - that is, an attacker who can only observe public data cannot learn anything about secret data.
To provide a better intuition lets consider two runs of the same program in which we have the *same public inputs* but *different secret values*. We consider a program to be non-interfering if the observable outputs of both runs are the same. If the observable outputs differ, then we can conclude that information about the secret data has influenced the public output, and hence there is interference. A simple example if we go back to the first example in the motivation

```
x   = SECRET   # some secret
y   = x
OBS = y        # observable data
```

Lets consider $x = 3$ and $x = 5$, because we have a direct flow from the secrets to the observable data, we get different observable outputs $y = 3$ and $y = 5$. Hence, we can conclude that there is interference as the attacker can distinguish between the two runs.
So the idea in short is that a program is non-interfering if whenver two intial states differ only in their secret variables, the resulting observed states are the same.

### 1.1.3 The formal definition

Let's consider our classical operational semantics

$$\langle P, \sigma \rangle \to S'$$

where $P$ is a program, $\sigma$ is the initial state and $\sigma'$ is the resulting state after executing $P$ in state $\sigma$. We define the restriction or slide of $\sigma$'s domain to the observable variables as $\sigma \downarrow \text{obs}$.

So a simple example, given

$$\sigma = \{\text{obs} \mapsto 4; \text{sec} \mapsto 5\} \text{ then } \sigma \downarrow \text{obs} = \{\text{obs} \mapsto 4\}$$

**Definition 1.1.** Non-interference
Formally we can now define non-interference as follows:

$$\forall \sigma_L, \sigma_R.\ \text{if } (\sigma_L \downarrow \text{obs}) \wedge (\langle s, \sigma_L \rangle \Downarrow \sigma'_L) \wedge (\langle s, \sigma_R \rangle \Downarrow \sigma'_R) \text{ then } (\sigma'_L \downarrow \text{obs} = \sigma'_R \downarrow \text{obs})$$

where $\sigma_L$ and $\sigma_R$ are two initial states that may differ in their secret variables but are the same in their observable variables. After executing the program $s$ in both states, we get resulting states $\sigma'_L$ and $\sigma'_R$. The program is non-interfering if the observable parts of the resulting states are equal.

**Example 1.1.** Non-interference example Let's consider the following program, and ask if it is non-interfering:

```
OBS = OBS + 1
```

To check for non-interference, we consider two initial states $\sigma_L$ and $\sigma_R$ that differ only in their secret variables. For example:

$$\sigma_L = \{\text{obs} \mapsto 4; \text{sec} \mapsto 5\}$$

$$\sigma_R = \{\text{obs} \mapsto 4; \text{sec} \mapsto 10\}$$

Both states have the same observable variable obs but different secret variables sec. Now, we execute the program in both states:

$$\langle s, \sigma_L \rangle \Downarrow \sigma'_L = \{\text{obs} \mapsto 5; \text{sec} \mapsto 5\}$$

$$\langle s, \sigma_R \rangle \Downarrow \sigma'_R = \{\text{obs} \mapsto 5; \text{sec} \mapsto 10\}$$

Now, we check the observable parts of the resulting states:

$$\sigma'_L \downarrow \text{obs} = \{\text{obs} \mapsto 5\}$$

$$\sigma'_R \downarrow \text{obs} = \{\text{obs} \mapsto 5\}$$

Since the observable parts of both resulting states are equal, we conclude that the program is non-interfering.

**Example 1.2.** Let's consider a more interesting example

```
x = SECRET
while (x >= 1):
    x = x + 1
```

Something which might be apparent here is that we have no explicit observable output. However, we can still have an information flow through the termination behaviour of the program. To check for non-interference, we consider two initial states $\sigma_L$ and $\sigma_R$ that differ only in their secret variables. For example:

$$\sigma_L = \{\text{sec} \mapsto 2\}$$

$$\sigma_R = \{\text{sec} \mapsto 5\}$$

Both states have different secret variables sec. Now, we execute the program in both states:

$$\langle s, \sigma_L \rangle \Downarrow \sigma'_L \text{ (terminates)}$$

$$\langle s, \sigma_R \rangle \Downarrow \sigma'_R \text{ (does not terminate)}$$

Now, we check the observable parts of the resulting states. However, since the program does not terminate in the second case, we cannot obtain a resulting state $\sigma'_R$. This means that the observable behavior of the program differs based on the secret input, as an attacker can observe whether the program terminates or not. Since the observable behavior (termination) differs based on the secret input, we conclude that the program is interfering, though importantly not by our naive definition.

### 1.1.4 Self-composition

One way to verify non-interference is through a technique called self-composition. The idea is to transform the original program into a new program that simulates two runs of the original program with different secret inputs but the same public inputs. As an example we can once again consider our intial simple program

```
x   = SECRET   # some secret
y   = x
OBS = y        # observable data
```

We can self-compose this program as follows:

```
assume (OBS_1 == OBS_2)   # assume public inputs are the same
x1    = SECRET_1           # first secret
y1    = x1
OBS_1 = y1                # first observable data

x2    = SECRET_2           # second secret
y2    = x2
OBS_2 = y2                # second observable data
assert (OBS_1 == OBS_2)   # assert public outputs are the same
```

This is called self-composition or the product program.

# 2 Information Flow Types

## 2.1 Motivation

The overarching paradigm of the previous approaches to having secure programs is predicated on the idea of

1. A user writing a program
2. A system proving the correctness of said program, i.e. verification.

However this is not the only way of ensuring program correctness. Another idea takes a more *proactive* approach, where instead of verifying a program after it has been written, we instead restrict the ways in which a program can be written in the first place. In simple terms, we cannot even write insecure programs to begin with.
A natural drawback or risk with this approach is that we might rule out programs which are correct but less obviously so. A classic example of a language which at least in part takes this approach is Rust with its ownership system. Within the domain of security this paradigm is called *language-based security*.
The primary mechanism which enables this proactive design in verification is the use of *type systems*. For our circumstances we will define a type system for the Nano language such that if a program type checks, then it is guaranteed to be non-interfering.

## 2.2 Security Lattice

An important concept is that the two levels of security $\textsc{sec}$ and $\textsc{obs}$ form a lattice. The implication of this being that we can define a partial order - a relation which is reflexive, transitive and antisymmetric - over the security levels. In this context we would say that

$$a \sqsubseteq b \iff b \text{ is at least as confidential as } a$$

in other words we can understand $b$ as requiring higher (or equal) security clearance than $a$. The natural question would then be what ordering do we have over our two security levels? The answer is

$$\textsc{obs} \sqsubseteq \textsc{sec}$$

a good follow-up question would then be how do we define the join operation $\sqcup$ over our security levels? The answer is as follows:

$$(\textsc{sec} \sqcup \textsc{obs}) \triangleq \textsc{sec}$$

## 2.3 Typing

### 2.3.1 Typing context

We define a typing context $\Gamma$ as a mapping from variables to their security types, i.e.

$$\Gamma : \text{Var} \rightarrow \{\textsc{obs}, \textsc{sec}\}$$

**Example 2.1.** An safe program
As an example lets consider the following code snippet

```
x = SECRET    # {x ↦ SEC; SECRET ↦ SEC}
y = x         # {x ↦ SEC; y ↦ SEC; SECRET ↦ SEC}
OBS = 3       # {x ↦ SEC; y ↦ SEC; OBS ↦ OBS; SECRET ↦ SEC}
```

Here each line we annotated how we extended the typing context $\Gamma$. The final typing context would hence be

$$\Gamma = \{x \mapsto \textsc{sec}; y \mapsto \textsc{sec}; \textsc{obs} \mapsto \textsc{obs}; \textsc{secret} \mapsto \textsc{sec}\}$$

Something to observe here is that at no point do we have some kind of conflict with our mappings, i.e. we never try to assign a variable to a security type which is lower than its current type in the context. Hence this program type checks and is secure.

**Example 2.2.** An unsafe program
Now lets consider the case of our classically unsafe program

```
x = SECRET    # {x ↦ SEC; SECRET ↦ SEC}
y = x         # {x ↦ SEC; y ↦ SEC; SECRET ↦ SEC}
OBS = y       # {x ↦ SEC; y ↦ SEC; OBS ↦ OBS; SECRET ↦ SEC}
```

Here again we annotate how we extend the typing context $\Gamma$ at each line. The final typing context would hence be

$$\Gamma = \{x \mapsto \textsc{sec}; y \mapsto \textsc{sec}; \textsc{obs} \mapsto \textsc{obs}; \textsc{secret} \mapsto \textsc{sec}\}$$

However, at the last line when we try to assign $y$ to $\textsc{obs}$, we have a conflict since $y$ is currently mapped to $\textsc{sec}$ in the typing context, and $\textsc{sec} \not\sqsubseteq \textsc{obs}$. Hence this program does not type check and is insecure.

### 2.3.2 Typing rules

To formalize the notions from the previous section we introduce a set of typing rules / judgements for expressions and statements in our Nano language.

$$\text{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{T-Num}\ \frac{}{\Gamma \vdash n : \textsc{obs}} \qquad \text{T-Plus}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 + e_2 : \tau_1 \sqcup \tau_2} \qquad \text{T-}\top\ \frac{}{\Gamma \vdash e : \top} \qquad \text{T-Or}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \vee e_2 : \tau_1 \sqcup \tau_2}$$

$$\text{T-Eq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 = e_2 : \tau_1 \sqcup \tau_2}$$

here $\tau_1, \tau_2 \in \{\textsc{obs}, \textsc{sec}\}$

**Example 2.3.** Using simple typing judgments
Let's consider the following typing context

$$\Gamma = \{x \mapsto \textsc{obs};\ y \mapsto \textsc{sec}\}$$

Now we can answer simple questions using our typing judgments:

- *Can we derive that* $\Gamma \vdash x : \textsc{obs}$ *?*
  Yes, by applying the T-Var rule we see that since $\Gamma(x) = \textsc{obs}$, we can conclude that $\Gamma \vdash x : \textsc{obs}$.

- *Can we derive that* $\Gamma \vdash x : \textsc{sec}$ ?
  No, because $\Gamma(x) = \textsc{obs}$ and there is no rule that allows us to derive $\Gamma \vdash x : \textsc{sec}$ from $\Gamma(x) = \textsc{obs}$.

Generally we can use typing judgements to determine the security types of variables and expressions based on the typing context $\Gamma$. Some more complex instances

$$\Gamma \vdash 3 : \tau \implies \tau = \textsc{obs} \qquad \text{(by T-Num)}$$
$$\Gamma \vdash x + 3 : \tau \implies \tau = \textsc{obs} \qquad \text{(by T-Plus and T-Num)}$$
$$\Gamma \vdash y + 3 : \tau \implies \tau = \textsc{sec} \qquad \text{(by T-Plus and T-Num)}$$
$$\Gamma \vdash x + y : \tau \implies \tau = \textsc{sec} \qquad \text{(by T-Plus)}$$

Importantly for the 3rd and 4th example we see how the security type of the expression is influenced by the highest security type of its components. That is, we apply the join operation $\sqcup$.

$$\textsc{obs} \sqcup \textsc{sec} \triangleq \textsc{sec}$$

### 2.3.3 Issue: Indirect Flows

Lets go back to our earlier example of an indirect flow

```
x = SECRET        # some secret data
OBS = 0           # public data
if x == 1:        # control flow depends on secret
    OBS = 1       # observable data (e.g. by attackers)
```

We know from earlier that this program does indeed have an information flow from SECRET to OBS through the control flow of the program. However, if we try to type check this program we get the following typing context at each line

1. $\{x \mapsto \textsc{sec}; \textsc{secret} \mapsto \textsc{sec}\}$
2. $\{x \mapsto \textsc{sec}; \textsc{obs} \mapsto \textsc{obs}; \textsc{secret} \mapsto \textsc{sec}\}$
3. $\{x \mapsto \textsc{sec}; \textsc{obs} \mapsto \textsc{obs}; \textsc{secret} \mapsto \textsc{sec}\}$
4. $\{x \mapsto \textsc{sec}; \textsc{obs} \mapsto \textsc{obs}; \textsc{secret} \mapsto \textsc{sec}\}$

Here we see that there is no conflict in the typing context, and hence this program type checks. This is a problem since we know this program is insecure. The issue here is that our current type system does not account for indirect flows through control flow.

### 2.3.4 Solution: Program Counter

To solve the issue of indirect flows being type checked we introduce the notion of a *program counter* PC which *tracks* the security label of the program counter. Two key instances where we employ this are

- CONDITIONALS/LOOPS: Here if we're in an if-statement or a loop which depends on a SEC variable then the program counter label is raised to SEC within the body of the conditional/loop.
- ASSIGNMENTS: Here if we assgn to a variable we need to check it is at least as confidential as the program counter.

### 2.3.5 Program Counter typing rules

$$\textsc{T-pc}\ \frac{}{\Gamma \vdash \textsc{pc} : s} \qquad \textsc{T-skip}\ \frac{}{\Gamma, \textsc{pc} \vdash \text{skip}} \qquad \textsc{T-Assign}\ \frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash e : \tau_2 \quad \tau_2 \sqsubseteq \tau_1 \quad \textsc{pc} \sqsubseteq \tau_1}{\Gamma, \textsc{pc} \vdash x := e} \qquad \textsc{T-Seq}\ \frac{\Gamma, \textsc{pc} \vdash s_1 \quad \Gamma, \textsc{pc} \vdash s_2}{\Gamma, \textsc{pc} \vdash s_1; s_2}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash b : \tau \quad \Gamma, \textsc{pc} \sqcup \tau \vdash s_1 \quad \Gamma, \textsc{pc} \sqcup \tau \vdash s_2}{\Gamma, \textsc{pc} \vdash \text{if } b \text{ then } s_1 \text{ else } s_2} \qquad \textsc{T-While}\ \frac{\Gamma \vdash b : \tau \quad \Gamma, \textsc{pc} \sqcup \tau \vdash s}{\Gamma, \textsc{pc} \vdash \text{while } b \text{ do } s}$$

$$\textsc{T-AppAsn}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1 \quad \textsc{pc} \sqsubseteq \tau_2}{\Gamma, \textsc{pc} \vdash f\ e} \qquad \textsc{T-ArrAsn}\ \frac{\Gamma \vdash a : \tau \quad \Gamma \vdash e_1 : \textsc{obs} \quad \Gamma \vdash e_2 : \tau' \quad \tau' \sqsubseteq \tau \quad \textsc{pc} \sqsubseteq \tau}{\Gamma, \textsc{pc} \vdash a[e_1] := e_2}$$

**Example 2.4.** Applying hardened typing rules
Let's again consider a typing context

$$\Gamma = \{x \mapsto \textsc{obs};\ y \mapsto \textsc{sec}\}$$

Now lets look at a few different typed expressions and see if they type check under the hardened typing rules:

- *Does* $\Gamma, \textsc{obs} \vdash x := y$ *type check?*
  **No**, because by applying the T-Assign rule we see that $\Gamma \vdash y : \textsc{sec}$ and $\Gamma \vdash x : \textsc{obs}$. However, $\textsc{sec} \not\sqsubseteq \textsc{obs}$ hence this assignment does not type check.

- *Does* $\Gamma, \textsc{obs} \vdash y := x$ *type check?*
  **Yes**, because by applying the T-Assign rule we see that $\Gamma \vdash x : \textsc{obs}$ and $\Gamma \vdash y : \textsc{sec}$. Here we have $\textsc{obs} \sqsubseteq \textsc{sec}$ and also $\textsc{obs} \sqsubseteq \textsc{sec}$ hence this assignment type checks.

- *Does* $\Gamma, \textsc{sec} \vdash x := 3$ *type check?*
  **No** because applying the T-Assign rule we see that $\Gamma \vdash 3 : \textsc{obs}$ and $\Gamma \vdash x : \textsc{obs}$. Here while $\textsc{obs} \sqsubseteq \textsc{obs}$ holds, we have $\textsc{sec} \not\sqsubseteq \textsc{obs}$ hence this assignment does not type check.

- *Does* $\Gamma, \textsc{obs} \vdash x := 3$ *type check?*
  **Yes** because applying the T-Assign rule we see that $\Gamma \vdash 3 : \textsc{obs}$ and $\Gamma \vdash x : \textsc{obs}$. Here we have $\textsc{obs} \sqsubseteq \textsc{obs}$ and also $\textsc{obs} \sqsubseteq \textsc{obs}$ hence this assignment type checks.

- *Does* $\Gamma, \textsc{sec} \vdash y := x$ *type check?*
  **Yes**, because by applying the T-Assign rule we see that $\Gamma \vdash x : \textsc{obs}$ and $\Gamma \vdash y : \textsc{sec}$. Here we have $\textsc{obs} \sqsubseteq \textsc{sec}$ and also $\textsc{sec} \sqsubseteq \textsc{sec}$ hence this assignment type checks.

## 2.4 System properties and extensions

### 2.4.1 Termination and while loops

An earlier point mentioned was that the naive definition of non-interference does not account for termination behavior. A good question to then ask is if our type system accounts for this? By default the answer is no. To give an example consider the following typing context and program

```
Γ = {y ↦ SEC}
while (y >= 0): # dependence on a secret variable
    y = y + 1
```

Here we can see that $y$ is considered a secret, and quite evidently the termination behavior of the program clearly depends on the value of $y$. So for example $y = 0$ would terminate while $y = 5$ would not, hence an attacker could learn information about $y$ through the termination behavior of the program.
The most obvious way to fix this issue is to simply make loop conditions only depend on $\textsc{obs}$ variables. In other words we disallow loops depending on secret variables entirely, hence we never have termination leaks. In our typing rule for while loops T-While we can enforce this by requiring that the condition $b$ type checks to $\textsc{obs}$.

$$\textsc{T-While2}\quad \frac{\Gamma \vdash b : \textsc{obs} \qquad \Gamma, \textsc{pc} \sqcup \textsc{obs} \vdash s}{\Gamma, \textsc{pc} \vdash \text{while } b \text{ do } s}$$

### 2.4.2 General guarantees

Generally we can say that for a statement $s$, and some environment $\Gamma$

$$\Gamma, \textsc{obs} \vdash s \equiv s \text{ is non-interfering}$$

## 2.5 Timing and Cache

### 2.5.1 Timing example

Let's consider the following piece of code

```
x = SEC
if x >= 1:
    b = a
    d = b + c
    ... # some time-consuming operations
```

Let's assume the following typing context

$$\Gamma = \{x \mapsto \textsc{sec};\ a \mapsto \textsc{sec};\ b \mapsto \textsc{sec};\ c \mapsto \textsc{sec}; d \mapsto \textsc{sec}\}$$

Lets analyze our program

- *Does the program type check?*
  **Yes**, because all variables are mapped to SEC in the typing context, and hence there are no conflicts.
- *Is the program non-interferent?*
  **Yes**, because there are no observable outputs in the program, and hence an attacker cannot learn anything about the secret variable
- *Is the program secure against timing attacks?*
  **No**, this is the crux of the issue, despite not having a loop or observable outputs, the timing behavior of the program depends on the secret variable $x$. If $x < 1$, the time-consuming operations are skipped, leading to a shorter execution time. An attacker could potentially measure the execution time and infer information about the secret variable $x$ based on whether the time-consuming operations were executed or not.

### 2.5.2 A short array primer

Before we give some examples of cache attacks, lets look at some examples of array accesses and their typing rules. We assume the following typing context for all examples

$$\Gamma = \{a \mapsto \textsc{obs};\ x \mapsto \textsc{obs};\ y \mapsto \textsc{sec};\ b \mapsto \textsc{sec}\}$$

- *Does* $y = a[x]$ *type check?*
  **Yes**, by applying the T-ArrIdx rule we see that $\Gamma \vdash a[x] : \textsc{obs}$ since $\Gamma \vdash x : \textsc{obs}$ and $\Gamma \vdash a : \tau$ for some $\tau$. Hence the assignment type checks since $\textsc{obs} \sqsubseteq \textsc{sec}$.
- *Does* $b = a[x]$ *type check?*
  **Yes**, by applying the T-ArrIdx rule we see that $\Gamma \vdash a[x] : \textsc{obs}$ since $\Gamma \vdash x : \textsc{obs}$ and $\Gamma \vdash a : \tau$ for some $\tau$. Hence the assignment type checks since $\textsc{obs} \sqsubseteq \textsc{sec}$.
- *Does* $x = a[y]$ *type check?*
  **No**, by applying the T-ArrIdx rule we see that $\Gamma \vdash a[y] : \textsc{sec}$ since $\Gamma \vdash y : \textsc{sec}$ and $\Gamma \vdash a : \tau$ for some $\tau$. Hence the assignment does not type check since $\textsc{sec} \not\sqsubseteq \textsc{obs}$.
- *Does* $y = b[y]$ *type check?*
  **No**, but there is some nuance here. Superficially the assumption might be that yes it does indeed type check i.e. it is secure, as we are only working with secrets. The issue here becomes that the access pattern to $b$ depends on the secret variable $y$. If an attacker can observe the access pattern (e.g. through cache timing attacks), they could potentially infer information about the secret variable $y$. Hence this assignment does not type check.

The critical takeaway here being that

- SECRET BRANCHING - You should not branch on secret data, as this can lead to timing leaks.
- SECRET INDEXING - You should not index arrays using secret data, as this can lead to access pattern leaks.

### 2.5.3 Attacker models

To resolve the seeming contradiction of non-intereference saying something is secure and cache/timing attacks suggesting otherwise we introduce the notion of *attacker models*. The idea being that non-interference provides security guarantees against a certain class of attackers, while cache/timing attacks consider a more powerful attacker model. In other words the notion of security is always defined relative to an attacker model, i.e. you first consider the capabilties of an attacker and then prove security against that model.

### 2.5.4 Cache-Timing attacker model

$$\textsc{T-If2}\ \frac{\Gamma \vdash b : \textsc{obs} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \text{if } b \text{ then } s_1 \text{ else } s_2} \qquad\qquad \textsc{T-ArrIdx}\ \frac{\Gamma \vdash e : \textsc{obs} \qquad \Gamma \vdash a : \tau}{\Gamma \vdash a[e] : \tau}$$

```
x = a[i₁]
y = a[i₂]
z = x + y
_ = z
```

$$\begin{tikzcd}
 & a[i_1] \arrow[r] & x \arrow[dr] & & & \\
\mathrm{src} \arrow[ur] \arrow[dr] & & & x + y \arrow[r] & z \arrow[r] & \mathrm{sink} \\
 & a[i_2] \arrow[r] & y \arrow[ur] & & &
\end{tikzcd}$$

## 2.6 Refinement Types

The simple idea of a refinement type system is to extend our basic type system with logical predicates which further constrain the values that a variable of a given type can take. As a simple example say we want to define a type for natural numbers. We can define a refinement type for natural numbers as follows:

$$\underbrace{\text{type}\ \mathbb{N} \triangleq \text{int}}_{\text{type alias}} \quad \underbrace{\{\nu \mid \nu \geqslant 0\}}_{\text{refinement predicate}}$$

The refinement acts as a restriction on the values inhabitants of a type make take. Some more complex examples

```
size :: x : array(α) → nat{ν : ν = length(x)}
get  :: x : array(α) → nat{ν : ν < length(x)} → α
```

There are a few key components here worth breaking down

- Dependent type - We can notice here that both in the signature of size and get the refinement predicate has a dependency on the argument of the function. In other words it's a value or runtime dependent type.
- Polymorphism - We can also notice that both functions are polymorphic over the type variable $\alpha$. This means that the functions can operate over arrays of any type.

We can see that the intuition of the refinements we are applying here should be pretty straightforward. In the case of size we are saying that the return value $\nu$ should be equal to the length of the input array $x$. In the case of get we are saying that the input index $\nu$ should be less than the length of the input array $x$, ensuring that we do not have out-of-bounds accesses.

**Example 2.5.** Refinement type function
We can also apply refinement types to functions as follows

$$(\lambda x.\ \text{add}\ x\ 1) : [x : \text{nat} \to \text{int}\{\nu \mid x < \nu \wedge \nu > 0\}]$$

Here we define a lambda function which takes a natural number $x$ and returns its increment by 1. The refinement type specifies that for any input $x$ of type nat, the return value $\nu$ must be larger than the input $x$ and also greater than 0.

### 2.6.1 Grammar

The grammar to express refinement types will be based on the simply typed lambda calculus.

| | | | | |
|---|---|---|---|---|
| Terms | $e$ | $::=$ | $c$ | *constant* |
| | | $\mid$ | $x$ | *variable* |
| | | $\mid$ | $\text{let } x = e_1 \text{ in } e_2$ | *definition* |
| | | $\mid$ | $\lambda x.\ e$ | *abstraction* |
| | | $\mid$ | $e_1\ e_2$ | *application* |
| | | $\mid$ | $e\colon \tau$ | *type ascription* |
| Base Type | $b$ | $::=$ | $\text{int}$ | *integers* |
| Refinement | $r$ | $::=$ | $\{v \mid p\}$ | *predicate over value* $v$ |
| Type | $\tau$ | $::=$ | $b\{r\}$ | *refined base type* |
| | | $\mid$ | $x\colon \tau_1 \to \tau_2$ | *dependent function type* |
| Typing Context | $\Gamma$ | $::=$ | $\cdot$ | *empty context* |
| | | $\mid$ | $\Gamma, x\colon \tau$ | *extended context* |

### 2.6.2 Typing Judgements

Entailment

Ent-Emp
$$\frac{\text{SMTValid}(c)}{\varnothing \vdash c}$$

Ent-Ext
$$\frac{\Gamma \vdash \forall x : b.\ p \Rightarrow c}{\Gamma, x : b\{x : p\} \vdash c}$$

Subtyping

Sub-Base
$$\frac{\Gamma \vdash \forall v_1 : b.\ p_1 \Rightarrow p_2[v_1 \mapsto v_2]}{\Gamma \vdash b\{v_1 : p_1\} <: b\{v_2 : p_2\}}$$

Sub-Fun
$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma, x_2 : \tau_2 \vdash t_1[x_2 \mapsto x_1] <: t_2}{\Gamma \vdash x_1 : \tau_1 \to t_1 <: x_2 : \tau_2 \to t_2}$$

Synthesis

$$\begin{aligned} \text{prim}(0) &= \text{int}\{v \mid v = 0\} \\ \text{prim}(1) &= \text{int}\{v \mid v = 1\} \\ \text{prim}(+) &= x : \text{int} \to y : \text{int} \to \text{int}\{v \mid v = x + y\} \end{aligned}$$

Syn-Var
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \Rightarrow \tau}$$

Syn-Con
$$\frac{\text{prim}(c) = \tau}{\Gamma \vdash c \Rightarrow \tau}$$

Syn-App
$$\frac{\Gamma \vdash e_1 \Rightarrow x : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2[x \mapsto e_2]}$$

Syn-Ann
$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}$$

Checking

Chk-Sym
$$\frac{\Gamma \vdash e \Rightarrow \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash e \Leftarrow \tau}$$

Chk-Lam
$$\frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\ e \Leftarrow x : \tau_1 \to \tau_2}$$

Chk-Let
$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Leftarrow \tau_2}$$