

WHAT TYPES ARE GOOD FOR

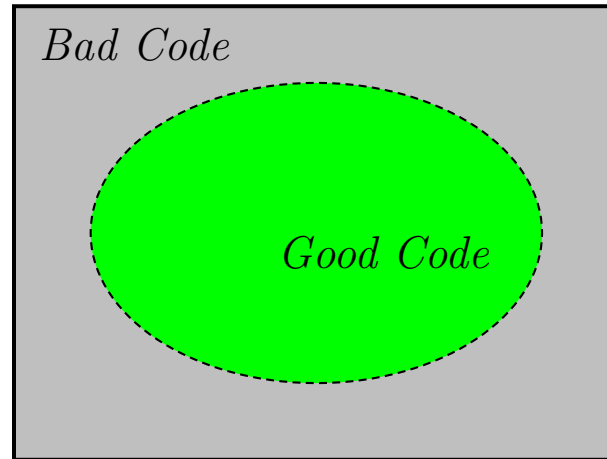
Assaf Kfoury

February 27, 2003

There are 3 parts in this slide presentation:

1. A general non-technical introduction to type systems for programming languages, entitled “What types are good for”, pp 1-21.
2. A technical presentation of a type system, the so-called system of *simple types*, for a small functional language — or the applicative core of any programming language. This part is entitled “Programming with safety”, pp 22-28.
3. A technical presentation of the system of simple types, now annotated with *security properties*, for the same functional language considered in the second part of the presentation. This third part is entitled “Programming with secrecy and integrity”, pp 29-46, based on a paper with the same title by N. Heintze and J. Riecke which appeared in the proceedings of POPL 1998.

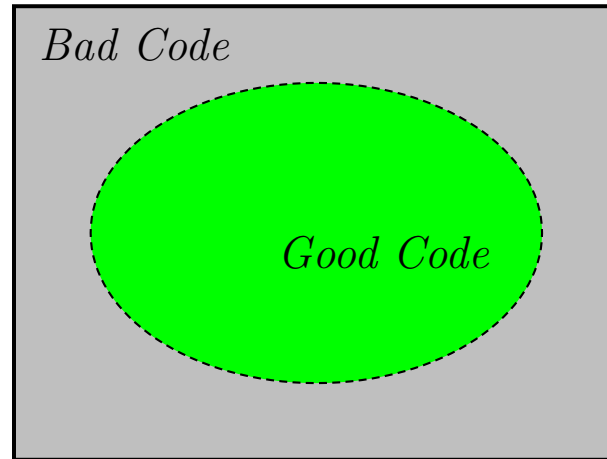
Writing code in your favorite programming language ...



Good Code =

- can be compiled,
- ...

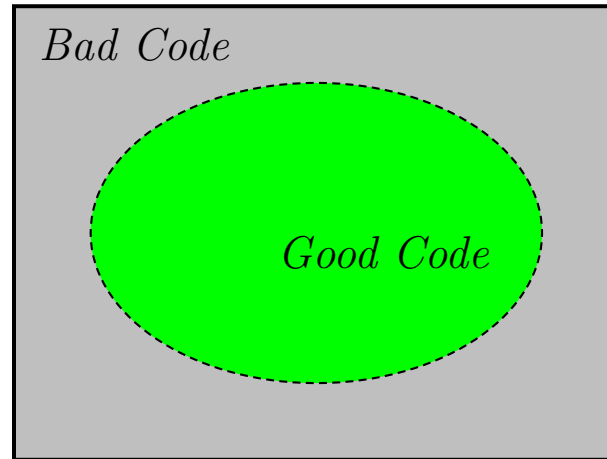
Writing code in your favorite programming language ...



Good Code =

- can be compiled,
- never passes an argument to a function that **does not know how to use it**,
- ...

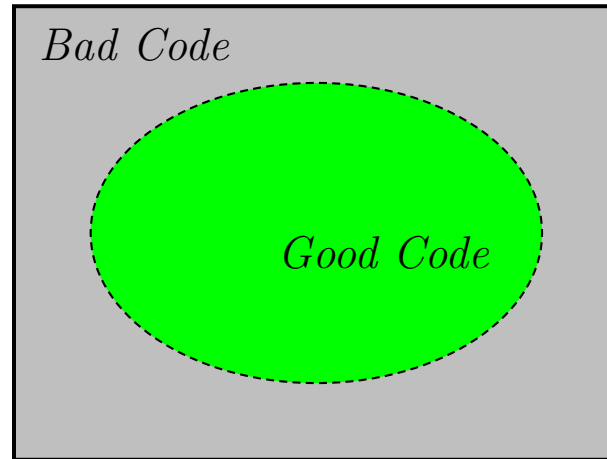
Writing code in your favorite programming language ...



Good Code =

- can be compiled,
- never passes an argument to a function that **does not know how to use it**,
- never passes an argument to a function that **writes it to allocated memory**, even if it knows how,
- ...

Writing code in your favorite programming language ...



Good Code =

- can be compiled,
- never passes an argument to a function that **does not know how** to use it,
- never passes an argument to a function that **writes it to allocated memory**, even if it knows how,
- never passes an argument to a function that **is not authorized** to use it, even if it knows how,
- ...

Other possible requirements of good code:

- management of resources,
- quality of service,
- ...

- Can we enforce these requirements at run time?
Often yes, but at a cost, sometimes severe.*

*Up to 70 percent of CPU time is spent on dynamic type-checking.

- Can we enforce these requirements at run time?
Often yes, but at a cost, sometimes severe.*
- Instead, can we guarantee these requirements **before** run time?
Yes, also at a cost, but much smaller, which is . . .

*Up to 70 percent of CPU time is spent on dynamic type-checking.

- Can we enforce these requirements at run time?
Often yes, but at a cost, sometimes severe.*
 - Instead, can we guarantee these requirements **before** run time?
Yes, also at a cost, but much smaller, which is
- type systems for programming languages!**

*Up to 70 percent of CPU time is spent on dynamic type-checking.

- Can we enforce these requirements at run time?
Often yes, but at a cost, sometimes severe.*
- Instead, can we guarantee these requirements **before** run time?
Yes, also at a cost, but much smaller, which is

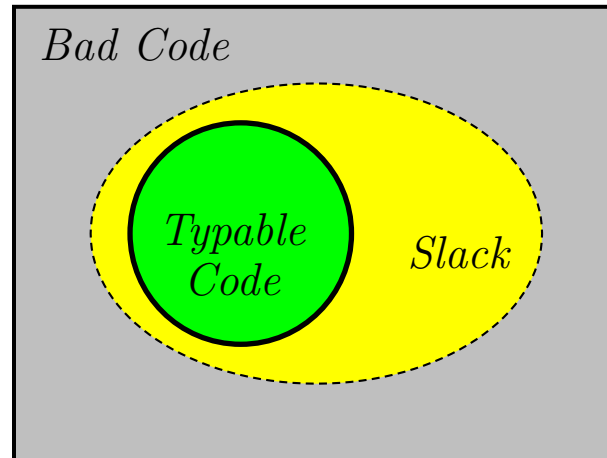
type systems for programming languages!

- As the trend continues towards high-level programming features — objects, module systems, first-class higher-order functions, . . . :

How much do we — and how much can we — shift the burden from run-time to compile-time?

*Up to 70 percent of CPU time is spent on dynamic type-checking.

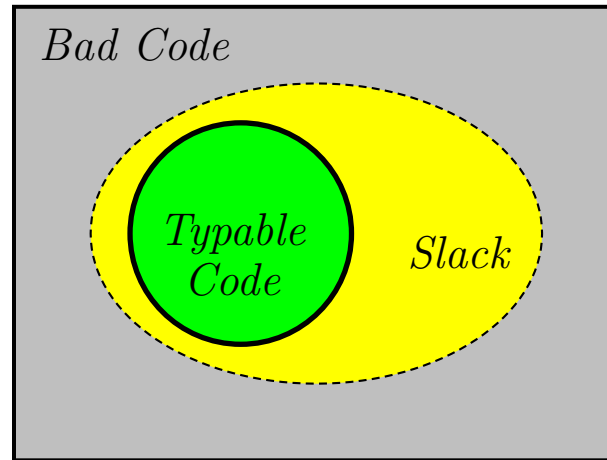
A **type system** for your favorite programming language imposes **invariants** on all executions of the same code.



Typable Code = good code that never breaks type-specified **invariants**.

Slack = good code that may break type-specified **invariants**.

A **type system** for your favorite programming language imposes **invariants** on all executions of the same code.



Typable Code = good code that never breaks type-specified **invariants**.

Slack = good code that may break type-specified **invariants**.

The *slack* contains all code that is unfairly rejected. The need to minimize it is one of the driving forces in the development of new type systems. But the new type systems can become unduly complex — with many bad consequences.

Desirable properties of a type system:

- Types are “easy” to write and insert into programs.
- Types are “easy” to understand.
- Efficient type-checking (for fully type-annotated code)
 - in very low-degree polynomial time,
on the average if not always.
- Efficient type-inference (for untyped/partially typed code)
 - in very low-degree polynomial time,
on the average if not always.

Desirable properties of a type system:

- Types are “easy” to write and insert into programs.
- Types are “easy” to understand.
- Efficient type-checking (for fully type-annotated code)
 - in very low-degree polynomial time,
on the average if not always.
- Efficient type-inference (for untyped/partially typed code)
 - in very low-degree polynomial time,
on the average if not always.

Two different uses of a type system:

- **Program specification.**
- **Program analysis.**

Other issues:

- Polymorphism.
- Compositionality.
- More program features: modules, objects, streams, ...
- Type-directed compilation.
- Type isomorphism.
- ...

What is the guarantee that a type system is not “leaking”?

What is the guarantee that a type system is not “leaking”?

Safety.

What is the guarantee that a type system is not “leaking”?

Safety.

Single most fundamental requirement on a type system, taking an **operational** approach to program semantics:

Safety = Progress + Preservation.*

*Many take **Safety = Soundness.**

What is the guarantee that a type system is not “leaking”?

Safety.

Single most fundamental requirement on a type system, taking an **operational** approach to program semantics:

$$\mathbf{Safety = Progress + Preservation.*}$$

Progress = a well-typed expression is not stuck, i.e., either it is a final value, or it raises an exception, or it can be evaluated further.

*Many take **Safety = Soundness.**

What is the guarantee that a type system is not “leaking”?

Safety.

Single most fundamental requirement on a type system, taking an **operational** approach to program semantics:

$$\mathbf{Safety} = \mathbf{Progress} + \mathbf{Preservation}.*$$

Progress = a well-typed expression is not stuck, i.e., either it is a final value, or it raises an exception, or it can be evaluated further.

Preservation = if a well-typed expression is evaluated one more step, the resulting expression is also well typed.[†]

*Many take **Safety** = **Soundness**.

†**Preservation** is usually further qualified.

What is the guarantee that a type system is not “leaking”?

Safety.

Single most fundamental requirement on a type system, taking an **operational** approach to program semantics:

$$\mathbf{Safety} = \mathbf{Progress} + \mathbf{Preservation}.*$$

Progress = a well-typed expression is not stuck, i.e., either it is a final value, or it raises an exception, or it can be evaluated further.

Preservation = if a well-typed expression is evaluated one more step, the resulting expression is also well typed.[†]

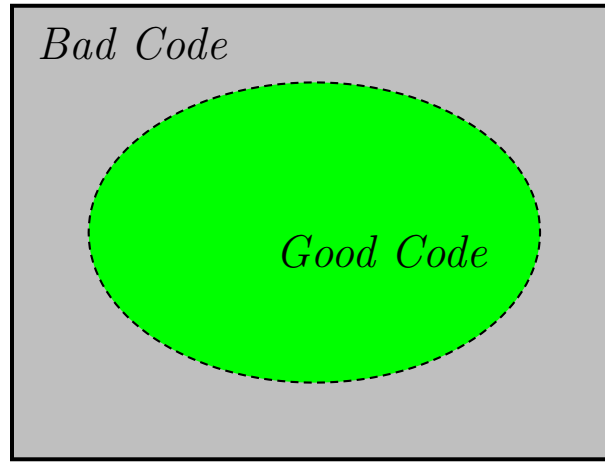
Robin Milner: “Well-typed programs do not go wrong.”

*Many take **Safety** = **Soundness**.

†**Preservation** is usually further qualified.

PROGRAMMING WITH SAFETY

(“No function is ever applied to the wrong argument”)



Good Code =

- can be compiled, and
- never passes an argument to a function that **does not know how** to use it.

Syntax of types:

$t ::= \text{bool} \mid \text{int} \mid \dots$	type constants
$\mid (t \rightarrow t')$	function types
$\mid (t \times t')$	product types
$\mid (t + t')$	sum types
$\mid \dots$	(other types according to need)

Syntax of values:

$v ::= \text{true} \mid \text{false}$	boolean
$\mid \dots \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$	integer
$\mid \langle v, v' \rangle$	pair of values
$\mid (\lambda x : t. e)$	abstraction
$\mid \dots$	(other values according to need)

Syntax of terms:

$e ::= x$	variable
$\mid v$	value
$\mid \langle e, e' \rangle$	pair of terms
$\mid (e e')$	application
$\mid (\text{if } e \text{ then } e' \text{ else } e'')$	conditional
$\mid \dots$	

Operational semantics:

$$((\lambda x : t. e) v) \Rightarrow e[x := v]$$

$$\text{proj}_i \langle v_1, v_2 \rangle \Rightarrow v_i$$

$$\text{if true then } e_1 \text{ else } e_2 \Rightarrow e_1$$

$$\text{if false then } e_1 \text{ else } e_2 \Rightarrow e_2$$

$$\dots \Rightarrow \dots$$

Typing Rules:

Var $\Gamma, x : t \vdash x : t$

Int $\Gamma \vdash n : \mathbf{int}$

Bool $\Gamma \vdash b : \mathbf{bool}$

Abs
$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1. e) : (t_1 \rightarrow t_2)}$$

App
$$\frac{\Gamma \vdash e : (t_1 \rightarrow t_2) \quad \Gamma \vdash e' : t_1}{\Gamma \vdash (e e') : t_2}$$

Pair
$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (t_1 \times t_2)}$$

Proj
$$\frac{\Gamma \vdash e : (t_1 \times t_2)}{\Gamma \vdash (\mathbf{proj}_i e) : t_i}$$

...

...

Theorem (Progress)

Suppose e is a closed well-typed expression, i.e., $\vdash e : t$ for some type t .

Then either e is a value or else there is some e' such that $e \Rightarrow e'$.

Theorem (Progress)

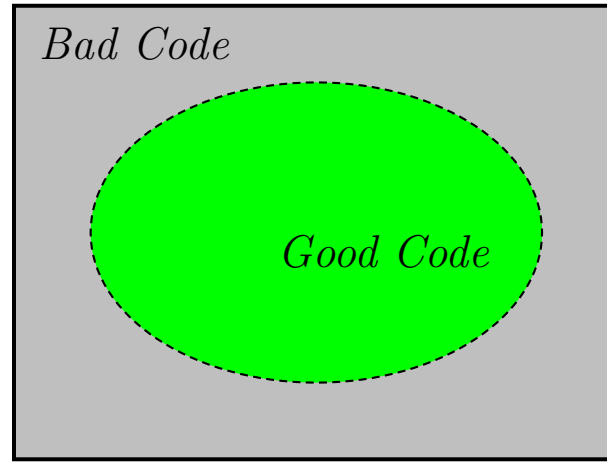
Suppose e is a closed well-typed expression, i.e., $\vdash e : t$ for some type t .
Then either e is a value or else there is some e' such that $e \Rightarrow e'$.

Theorem (Preservation)

If $\Gamma \vdash e : t$ and $e \Rightarrow e'$ then $\Gamma \vdash e' : t$.

PROGRAMMING WITH SECRECY AND INTEGRITY

(based on a paper by N. Heintze and J. Riecke)



Good Code =

- can be compiled,
- never passes an argument to a function that **does not know how** to use it,
- never passes an argument to a function that **is not authorized** to use it, even if it knows how.

GENERAL GOALS:

- Associate security with **data objects**, not locations.
(Alternative: **secure persistent store** that controls access to each location.)
- Trace security through information flow.
- Security level does not decrease from input to output.
- Distinguish between **secrecy** (who finds out about X) and **integrity** (who is responsible for X).
- Finer-grained analysis: Associate more than one security level with each X .

Different security attributes attached to each X :

direct readers, direct creators,
indirect readers, indirect creators.

Different security attributes attached to each X :

direct readers, direct creators,
indirect readers, indirect creators.

Example:

$$P = \mathbf{if} (x > 25) \mathbf{then} y := 1 \mathbf{else} y := 2$$

- A has read-access to P and y only.
- B has write-access to x only.
- C has write-access to x and can execute P .

Different security attributes attached to each X :

direct readers, direct creators,
indirect readers, indirect creators.

Example:

$$P = \mathbf{if} (x > 25) \mathbf{then} y := 1 \mathbf{else} y := 2$$

- A has read-access to P and y only.
 $A =$ indirect reader of x 's content.
- B has write-access to x only.
 $B =$ indirect creator of y 's content.
- C has write-access to x and can execute P .
 $C =$ direct creator of y 's content.

In addition to its type, every X has 4 security attributes:

1. Who are its **direct readers** — the set r ,
2. Who are its **indirect readers** — the set ir ,
3. Who are its **direct creators** — the set c ,
4. Who are its **indirect creators** — the set ic .

Two invariants: $r \subseteq ir$ and $c \subseteq ic$.

(r, ir) specify X 's **secrecy**: **who finds out about X** .

(c, ic) specify X 's **integrity**: **who is responsible for X** .

Examples of annotated types:

\mathbb{U} = the universe of all users.

$\mathcal{P}(\mathbb{U})$ = the universe of all security groups.

Suppose there are 5 linearly ordered security groups:

$$\emptyset \subseteq H \subseteq M \subseteq L \subseteq \mathbb{U}$$

Examples of annotated types:

\mathbb{U} = the universe of all users.

$\mathcal{P}(\mathbb{U})$ = the universe of all security groups.

Suppose there are 5 linearly ordered security groups:

$$\emptyset \subseteq H \subseteq M \subseteq L \subseteq \mathbb{U}$$

(int, *M*, *L*, *H*, *M*)
r *ir* *c* *ic*

Examples of annotated types:

\mathbb{U} = the universe of all users.

$\mathcal{P}(\mathbb{U})$ = the universe of all security groups.

Suppose there are 5 linearly ordered security groups:

$$\emptyset \subseteq H \subseteq M \subseteq L \subseteq \mathbb{U}$$

$$\begin{array}{c} (\text{int}, M, L, H, M) \\ r \quad ir \quad c \quad ic \end{array}$$

$$((\text{int}, M, L, H, M) \times (\text{bool}, M, M, M, M), \mathbb{U}, \mathbb{U}, \emptyset, \emptyset)$$

Examples of annotated types:

\mathbb{U} = the universe of all users.

$\mathcal{P}(\mathbb{U})$ = the universe of all security groups.

Suppose there are 5 linearly ordered security groups:

$$\emptyset \subseteq H \subseteq M \subseteq L \subseteq \mathbb{U}$$

$$\begin{array}{c} (\text{int}, M, L, H, M) \\ r \quad ir \quad c \quad ic \end{array}$$

$$((\text{int}, M, L, H, M) \times (\text{bool}, M, M, M, M), \mathbb{U}, \mathbb{U}, \emptyset, \emptyset)$$

$$((t_1, \kappa_1) \rightarrow (t_2, \kappa_2), \kappa_3)$$

with $\kappa_i \in \mathcal{P}(\mathbb{U}) \times \mathcal{P}(\mathbb{U}) \times \mathcal{P}(\mathbb{U}) \times \mathcal{P}(\mathbb{U})$

Example

```
type user_ids = listof int
```

```
fun lookup ([ ]:user_ids) id = false:bool  
  | lookup ((x::xs):user_ids) id = if x = id  
                                   then true:bool  
                                   else lookup xs id
```

Example

```
type user_ids = listof int

fun lookup ([ ]:user_ids) id = false:bool
  | lookup ((x::xs):user_ids) id = if x = id
                                   then true:bool
                                   else lookup xs id
```

We want to allow users in security group M to check user id's,
but not to have free access to the entire list of user id's.

Only users in security group H have free access to the entire list of user id's.

Example

```
type user_ids = (listof (int,  $H, M$ ),  $M, M$ )

fun lookup ([ ]:user_ids) id = false:(bool,  $M, M$ )
  | lookup ((x::xs):user_ids) id = if x = id
                                   then true:(bool,  $M, M$ )
                                   else lookup xs id
```

We want to allow users in security group M to check user id's,
but not to have free access to the entire list of user id's.

Only users in security group H have free access to the entire list of user id's.

An appropriate type for the function `lookup` is

```
lookup: (user_ids -> (int,  $H, M$ ) -> (bool,  $M, M$ ),  $M, M$ )
```

Restrict security properties to direct/indirect readers only

Syntax of types and secure types:

$\kappa ::= (r, ir)$	security properties
$t ::= \mathbf{bool} \mid \mathbf{int} \mid \dots$	type constants
$\mid (s \rightarrow s')$	function types
$\mid (s \times s')$	product types
$\mid (s + s')$	sum types
$\mid \dots$	(other types according to need)
$s ::= (t, \kappa)$	secure types

Syntax of basic values and values:

$bv ::= \text{true} \mid \text{false}$	boolean
$\mid \dots \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$	integer
$\mid \langle v, v' \rangle$	pair of values
$\mid (\lambda x : s. e)$	abstraction
$\mid \dots$	(other basic values according to need)
$v ::= bv_\kappa$	value, i.e., a basic value with a security property

Syntax of terms:

$e ::= x$	variable
$\mid v$	value
$\mid \langle e, e' \rangle_\kappa$	pair of terms
$\mid (e e')_r$	application
$\mid (\text{if } e \text{ then } e' \text{ else } e'')_r$	conditional
$\mid (\text{protect}_r e)$	protected term (security of e is increased)
$\mid \dots$	

Operational semantics:

Given a security property $\kappa = (r, ir)$ and a security group r' , the expression $\kappa \bullet r'$ is the security property $\kappa' = (r \cap r', ir \cap r')$.

This operation is extended to values as follows. Given value bv_κ and security group r , the expression $bv_\kappa \bullet r$ denotes the “more secure” value $bv_{\kappa \bullet r}$.

$$\begin{aligned} ((\lambda x : s. e)_{r, ir} v)_{r'} &\Rightarrow (\text{protect}_{ir} e[x := v]) \quad \text{if } r \supseteq r' \\ (\text{proj}_i \langle v_1, v_2 \rangle_{r, ir})_{r'} &\Rightarrow (\text{protect}_{ir} v_i) \quad \text{if } r \supseteq r' \text{ and } i \in \{1, 2\} \\ (\text{if true}_{r, ir} \text{ then } e_1 \text{ else } e_2)_{r'} &\Rightarrow (\text{protect}_{ir} e_1) \quad \text{if } r \supseteq r' \\ (\text{if false}_{r, ir} \text{ then } e_1 \text{ else } e_2)_{r'} &\Rightarrow (\text{protect}_{ir} e_2) \quad \text{if } r \supseteq r' \\ \dots &\Rightarrow \dots \\ (\text{protect}_{ir} v) &\Rightarrow v \bullet ir \end{aligned}$$

Subtyping:

Define $(r_1, ir_1) \leq (r_2, ir_2)$ iff $r_1 \supseteq r_2$ and $ir_1 \supseteq ir_2$.

In words, there is an increase of security from (r_1, ir_1) to (r_2, ir_2)

The relation “ \leq ” extended to all types by induction:

$$\begin{array}{ccc} \frac{s_1 \leq s_2 \quad s_2 \leq s_3}{s_1 \leq s_3} & \frac{\kappa \leq \kappa'}{(\text{true}, \kappa) \leq (\text{true}, \kappa')} & \dots \\ \\ \frac{\kappa \leq \kappa' \quad s'_1 \leq s_1 \quad s_2 \leq s'_2}{(s_1 \rightarrow s_2, \kappa) \leq (s'_1 \rightarrow s'_2, \kappa')} & \frac{\kappa \leq \kappa' \quad s_1 \leq s'_1 \quad s_2 \leq s'_2}{(s_1 \times s_2, \kappa) \leq (s'_1 \times s'_2, \kappa')} & \dots \\ \\ \dots & \dots & \dots \end{array}$$

Typing Rules:

Var $\Gamma, x : s \vdash x : s$

Int $\Gamma \vdash n_\kappa : (\mathbf{int}, \kappa)$

Sub
$$\frac{\Gamma \vdash e : s \quad s \leq s'}{\Gamma \vdash e : s'}$$

Abs
$$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1. e)_\kappa : (s_1 \rightarrow s_2, \kappa)}$$

Pair
$$\frac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle_\kappa : (s_1 \times s_2, \kappa)}$$

...

Bool $\Gamma \vdash b_\kappa : (\mathbf{bool}, \kappa)$

Protect
$$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\mathbf{protect}_r e) : s \bullet r}$$

App
$$\frac{\Gamma \vdash e : (s_1 \rightarrow s_2, (r, ir)) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e e')_{r'} : s_2 \bullet ir} \quad r \supseteq r'$$

Proj
$$\frac{\Gamma \vdash e : (s_1 \times s_2, (r, ir))}{\Gamma \vdash (\mathbf{proj}_i e)_{r'} : s_i \bullet ir} \quad r \supseteq r'$$

...