

This note, extracted from a technical report [Bengtson et al., 2008], summarizes the definition of RCF, the  $\lambda$ -calculus underlying the F7 typechecker. Visit <http://research.microsoft.com/F7> for more information about the typechecker.

The  $\lambda$ -calculus RCF is an assembly of standard parts: call-by-value dependent functions, dependent pairs, sums, iso-recursive types, message-passing concurrency, refinement types, subtyping, and a universal type  $\text{Un}$  to model attacker knowledge. This is essentially the Fixpoint Calculus (FPC) [Gunter, 1992], augmented with concurrency and refinement types. Hence, we adopt the name Refined Concurrent FPC, or RCF for short.

Section 1 describes a class of logics that may be used to specify the constraints within refinement types. Section 2 describes the syntax and operational semantics of the calculus, as well as a notion of safety, that no assertions fail. Section 3 defines a type system suitable for proving expressions safe by typing. Section 4 introduces a stronger condition, robust safety, for expressing correctness properties of security protocols, together with a modified type system suitable for proving expressions robustly safe.

## 1 Authorization Logics

The calculus relies on logical formulas to specify correctness properties. These formulas are drawn from any choice of authorization logic, a logic satisfying the properties below. (In our initial implementation, the authorization logic is simply first-order logic with equality.)

An *authorization logic* is given as a set of *formulas* defined by a grammar that includes the one given below and a *deducibility relation*  $S \vdash C$ , from finite multisets of formulas to formulas that meets the properties listed below. (The set of values, ranged over by  $M$ , is defined in Section 2.)

### Minimal Syntax of Formulas:

$p$	predicate symbol
$C ::=$	formula
$p(M_1, \dots, M_n)$	atomic formula
$M = M'$	equation
$C \wedge C'$	conjunction
$C \vee C'$	disjunction
$\neg C$	negation
$\forall x.C$	universal quantification
$\exists x.C$	existential quantification

$$\text{True} \triangleq () = ()$$

$$\text{False} \triangleq \neg \text{True}$$

$$M \neq M' \triangleq \neg(M = M')$$

$$(C \Rightarrow C') \triangleq (\neg C \vee C')$$

$$(C \Leftrightarrow C') \triangleq (C \Rightarrow C') \wedge (C' \Rightarrow C)$$

### Properties of Deducibility: $S \vdash C$

$S, C$  stands for  $S, \{C\}$ ; in (Subst),  $\sigma$  ranges over substitutions of values for variables and permutations of names.

(Axiom)	(Mon)	(Subst)	(Cut)	
$\frac{}{C \vdash C}$	$\frac{S \vdash C}{S, C' \vdash C}$	$\frac{S \vdash C}{S\sigma \vdash C\sigma}$	$\frac{S \vdash C \quad S, C \vdash C'}{S \vdash C'}$	
(And Intro)	(And Elim)	(Or Intro)		
$\frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$	$\frac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$	$\frac{S \vdash C_i}{S \vdash C_0 \vee C_1} \quad i = 0, 1$		
(Eq)	(Ineq)	(Ineq Cons)		
$\frac{}{\emptyset \vdash M = M}$	$\frac{M \neq N}{\emptyset \vdash M \neq N}$	$\frac{h N = M \text{ for no } N}{\emptyset \vdash \forall x. h x \neq M}$		
	$\frac{fv(M, N) = \emptyset}{\emptyset \vdash M \neq N}$	$\frac{fv(M) = \emptyset}{\emptyset \vdash \forall x. h x \neq M}$		
(Exists Intro)	(Exists Elim)			
$\frac{S \vdash C\{M/x\}}{S \vdash \exists x.C}$	$\frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin fv(S, C')}{S \vdash C'}$			

FOL/F, which is first-order logic with the axiom schemas displayed below, is an example of an authorization logic. (The intended model consists of the phrases of syntax of RCF identified up to consistent renaming of bound names and variables. A *syntactic* function symbol is one used to represent the phrases of RCF as a term, using the locally nameless representation of de Bruijn. RCF variables are identified with the variables of the logic, while each RCF name is a constant, that is, a nullary syntactic function symbol.)

### Additional Rules for FOL/F:

(F Disjoint)	(F Injective)
$\frac{f \neq f' \text{ syntactic}}{S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) \neq f'(\vec{y})}$	$\frac{f \text{ syntactic}}{S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) = f(\vec{y}) \Rightarrow \vec{x} = \vec{y}}$

## 2 Expressions, Evaluation, and Safety

### Syntax of Values and Expressions:

$a, b, c$	name
$x, y, z$	variable
$h ::=$	value constructor
$\text{inl}$	left constructor of sum type
$\text{inr}$	right constructor of sum type
$\text{fold}$	constructor of recursive type
$M, N ::=$	value

$x$	variable
$()$	unit
$\mathbf{fun} x \rightarrow A$	function (scope of $x$ is $A$ )
$(M, N)$	pair
$h M$	construction
$A, B ::=$	expression
$M$	value
$M N$	application
$M = N$	syntactic equality
$\mathbf{let} x = A \mathbf{in} B$	let (scope of $x$ is $B$ )
$\mathbf{let} (x, y) = M \mathbf{in} A$	pair split (scope of $x, y$ is $A$ )
$\mathbf{match} M \mathbf{with}$	constructor match
$h x \rightarrow A \mathbf{else} B$	(scope of $x$ is $A$ )
$(\nu a)A$	restriction (scope of $a$ is $A$ )
$A \dot{\mapsto} B$	fork
$a!M$	transmission of $M$ on channel $a$
$a?$	receive message off channel
$\mathbf{assume} C$	assumption of formula $C$
$\mathbf{assert} C$	assertion of formula $C$
$\mathbf{true} \triangleq \mathbf{inl} ()$	$\mathbf{false} \triangleq \mathbf{inr} ()$

The formal syntax of expressions is in an intermediate, reduced form (reminiscent of A-normal form [Sabry and Felleisen, 1993]) where  $\mathbf{let} x = A \mathbf{in} B$  is the only construct to allow sequential evaluation of expressions. As usual,  $A; B$  is short for  $\mathbf{let} _ = A \mathbf{in} B$ . (The notation  $_$  denotes an anonymous variable that by convention occurs nowhere else.) More notably, if  $A$  and  $B$  are proper expressions rather than being values, the application  $A B$  is short for  $\mathbf{let} f = A \mathbf{in} (\mathbf{let} x = B \mathbf{in} f x)$ .

### Examples: Communication and Concurrency

$(T)\mathbf{chan} \triangleq (T \rightarrow \mathbf{unit}) \times (\mathbf{unit} \rightarrow T)$
$\mathbf{chan} \triangleq \mathbf{fun} _ \rightarrow (\nu a)(\mathbf{fun} x \rightarrow a!x, \mathbf{fun} _ \rightarrow a?)$
$\mathbf{send} \triangleq \mathbf{fun} c x \rightarrow \mathbf{let} (s, r) = c \mathbf{in} s x$ send $x$ on $c$
$\mathbf{recv} \triangleq \mathbf{fun} c \rightarrow \mathbf{let} (s, r) = c \mathbf{in} r ()$ block for $x$ on $c$
$\mathbf{fork} \triangleq \mathbf{fun} f \rightarrow (f()) \dot{\mapsto} ()$ run $f$ in parallel

### Structures and Static Safety:

$e ::= M \mid M N \mid M = N \mid \mathbf{let} (x, y) = M \mathbf{in} A \mid$ $\mathbf{match} M \mathbf{with} h x \rightarrow A \mathbf{else} B \mid a? \mid \mathbf{assert} C$
$\prod_{i \in 1..n} A_i \triangleq () \dot{\mapsto} A_1 \dot{\mapsto} \dots \dot{\mapsto} A_n$
$\mathcal{L} ::= \{ \} \mid (\mathbf{let} x = \mathcal{L} \mathbf{in} B)$
$\mathbf{S} ::= (\nu a_1) \dots (\nu a_\ell)$ $((\prod_{i \in 1..m} \mathbf{assume} C_i) \dot{\mapsto} (\prod_{j \in 1..n} c_j!M_j) \dot{\mapsto} (\prod_{k \in 1..o} \mathcal{L}_k\{e_k\}))$

Let structure  $\mathbf{S}$  be *statically safe* if and only if, for all  $k \in 1..o$  and  $C$ , if  $e_k = \mathbf{assert} C$  then  $\{C_1, \dots, C_m\} \vdash C$ .

Structures formalize the idea that a state has three parts: (1) the *log*, a multiset  $\prod_{i \in 1..m} \mathbf{assume} C_i$  of assumed formulas; (2) a series of messages  $M_j$  sent on channels but not

yet received; and (3) a series of elementary expressions  $e_k$  being evaluated in parallel contexts.

### Heating: $A \Rightarrow A'$

Axioms  $A \equiv A'$  are read as both  $A \Rightarrow A'$  and  $A' \Rightarrow A$ .

$A \Rightarrow A$	(Heat Refl)
$A \Rightarrow A''$ if $A \Rightarrow A'$ and $A' \Rightarrow A''$	(Heat Trans)
$A \Rightarrow A' \Rightarrow \mathbf{let} x = A \mathbf{in} B \Rightarrow \mathbf{let} x = A' \mathbf{in} B$	(Heat Let)
$A \Rightarrow A' \Rightarrow (\nu a)A \Rightarrow (\nu a)A'$	(Heat Res)
$A \Rightarrow A' \Rightarrow (A \dot{\mapsto} B) \Rightarrow (A' \dot{\mapsto} B)$	(Heat Fork 1)
$A \Rightarrow A' \Rightarrow (B \dot{\mapsto} A) \Rightarrow (B \dot{\mapsto} A')$	(Heat Fork 2)
$() \dot{\mapsto} A \equiv A$	(Heat Fork ())
$a!M \Rightarrow a!M \dot{\mapsto} ()$	(Heat Msg ())
$\mathbf{assume} C \Rightarrow \mathbf{assume} C \dot{\mapsto} ()$	(Heat Assume ())
$a \notin \mathbf{fn}(A') \Rightarrow A' \dot{\mapsto} ((\nu a)A) \Rightarrow (\nu a)(A' \dot{\mapsto} A)$	(Heat Res Fork 1)
$a \notin \mathbf{fn}(A') \Rightarrow ((\nu a)A) \dot{\mapsto} A' \Rightarrow (\nu a)(A \dot{\mapsto} A')$	(Heat Res Fork 2)
$a \notin \mathbf{fn}(B) \Rightarrow$ $\mathbf{let} x = (\nu a)A \mathbf{in} B \Rightarrow (\nu a)\mathbf{let} x = A \mathbf{in} B$	(Heat Res Let)
$(A \dot{\mapsto} A') \dot{\mapsto} A'' \equiv A \dot{\mapsto} (A' \dot{\mapsto} A'')$	(Heat Fork Assoc)
$(A \dot{\mapsto} A') \dot{\mapsto} A'' \Rightarrow (A' \dot{\mapsto} A) \dot{\mapsto} A''$	(Heat Fork Comm)
$\mathbf{let} x = (A \dot{\mapsto} A') \mathbf{in} B \equiv$ $A \dot{\mapsto} (\mathbf{let} x = A' \mathbf{in} B)$	(Heat Fork Let)

**Lemma 1 (Structure)** For every expression  $A$ , there is a structure  $\mathbf{S}$  such that  $A \Rightarrow \mathbf{S}$ .

### Reduction: $A \rightarrow A'$

$(\mathbf{fun} x \rightarrow A) N \rightarrow A\{N/x\}$	(Red Fun)
$(\mathbf{let} (x_1, x_2) = (N_1, N_2) \mathbf{in} A) \rightarrow$ $A\{N_1/x_1\}\{N_2/x_2\}$	(Red Split)
$(\mathbf{match} M \mathbf{with} h x \rightarrow A \mathbf{else} B) \rightarrow$ $\begin{cases} A\{N/x\} & \text{if } M = h N \text{ for some } N \\ B & \text{otherwise} \end{cases}$	(Red Match)
$M = N \rightarrow \begin{cases} \mathbf{true} & \text{if } M = N \\ \mathbf{false} & \text{otherwise} \end{cases}$	(Red Eq)
$a!M \dot{\mapsto} a? \rightarrow M$	(Red Comm)
$\mathbf{assert} C \rightarrow ()$	(Red Assert)
$\mathbf{let} x = M \mathbf{in} A \rightarrow A\{M/x\}$	(Red Let Val)
$A \rightarrow A' \Rightarrow \mathbf{let} x = A \mathbf{in} B \rightarrow \mathbf{let} x = A' \mathbf{in} B$	(Red Let)
$A \rightarrow A' \Rightarrow (\nu a)A \rightarrow (\nu a)A'$	(Red Res)
$A \rightarrow A' \Rightarrow (A \dot{\mapsto} B) \rightarrow (A' \dot{\mapsto} B)$	(Red Fork 1)
$A \rightarrow A' \Rightarrow (B \dot{\mapsto} A) \rightarrow (B \dot{\mapsto} A')$	(Red Fork 2)
$A \rightarrow A'$ if $A \Rightarrow B, B \rightarrow B', B' \Rightarrow A'$	(Red Heat)

A closed expression  $A$  is *safe* if and only if, in all evaluations of  $A$ , all assertions succeed.

### Expression Safety:

An expression  $A$  is *safe* if and only if, for all  $A'$  and  $\mathbf{S}$ , if  $A \rightarrow^* A'$  and  $A' \Rightarrow \mathbf{S}$ , then  $\mathbf{S}$  is statically safe.

### 3 A Type System for Safety

#### 3.1 Types and Environments

##### Syntax of Types:

$H, T, U, V ::= \text{type}$	
$\text{unit}$	unit type
$\Pi x : T. U$	dependent function type (scope of $x$ is $U$ )
$\Sigma x : T. U$	dependent pair type (scope of $x$ is $U$ )
$T + U$	disjoint sum type
$\mu \alpha. T$	iso-recursive type (scope of $\alpha$ is $T$ )
$\alpha$	iso-recursive type variable
$\{x : T \mid C\}$	refinement type (scope of $x$ is $C$ )

##### Some Derivable Types:

$\{C\} \triangleq \{- : \text{unit} \mid C\}$	(ok-type)
$\text{bool} \triangleq \text{unit} + \text{unit}$	
$\text{int} \triangleq \mu \alpha. \text{unit} + \alpha$	
$(T)\text{list} \triangleq \mu \alpha. \text{unit} + (T \times \alpha)$	
$T \rightarrow U \triangleq \Pi \_ : T. U$	
$[x_1 : T_1] \{C_1\} \rightarrow U \triangleq \Pi x_1 : \{x_1 : T_1 \mid C_1\}. U$	
$(x_1 : T_1 * \dots * x_n : T_n) \{C\} \triangleq$	
$\begin{cases} \Sigma x_1 : T_1. \dots \Sigma x_{n-1} : T_{n-1}. \{x_n : T_n \mid C\} & \text{if } n > 0 \\ \{C\} & \text{otherwise} \end{cases}$	

##### Syntax of Typing Environments:

$\mu ::=$	environment entry
$\alpha <: \alpha'$	subtype ( $\alpha \neq \alpha'$ )
$a \uparrow T$	name of a typed channel
$x : T$	variable
$E ::= \mu_1, \dots, \mu_n$	environment
$\text{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\}$	
$\text{dom}(a \uparrow T) = \{a\}$	
$\text{dom}(x : T) = \{x\}$	
$\text{dom}(\mu_1, \dots, \mu_n) = \text{dom}(\mu_1) \cup \dots \cup \text{dom}(\mu_n)$	
$\text{recvar}(E) = \{\alpha \mid \alpha \in \text{dom}(E)\}$	

#### 3.2 Typing Rules

The type system consists of five inductively defined judgments.

##### Judgments:

$E \vdash \diamond$	$E$ is syntactically well-formed
$E \vdash T$	in $E$ , type $T$ is syntactically well-formed
$E \vdash C$	formula $C$ is derivable from $E$
$E \vdash T <: U$	in $E$ , type $T$ is a subtype of type $U$
$E \vdash A : T$	in $E$ , expression $A$ has type $T$

##### Rules of Well-Formedness and Deduction:

(Env Empty)	(Env Entry)	(Type)
$\frac{}{\emptyset \vdash \diamond}$	$\frac{}{E \vdash \diamond}$	$\frac{}{E \vdash \diamond}$
	$\frac{\text{fnfv}(\mu) \subseteq \text{dom}(E)}{\text{dom}(\mu) \cap \text{dom}(E) = \emptyset}$	$\frac{}{\text{fnfv}(T) \subseteq \text{dom}(E)}$
	$\frac{}{E, \mu \vdash \diamond}$	$\frac{}{E \vdash T}$
(Derive)	$\frac{E \vdash \diamond \quad \text{fnfv}(C) \subseteq \text{dom}(E) \quad \text{forms}(E) \vdash C}{E \vdash C}$	
	$\text{forms}(E) \triangleq$	
	$\begin{cases} \{C\{y/x\}\} \cup \text{forms}(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ \text{forms}(E_1) \cup \text{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \emptyset & \text{otherwise} \end{cases}$	

##### General Rules:

(Sub Refl)	$\frac{E \vdash T \quad \text{recvar}(E) \cap \text{fnfv}(T) = \emptyset}{E \vdash T <: T}$	
(Val Var)	(Exp Subsum)	
$\frac{}{E \vdash \diamond} \quad (x : T) \in E$	$\frac{}{E \vdash A : T} \quad E \vdash T <: T'$	$\frac{}{E \vdash x : T} \quad E \vdash A : T'$
(Exp Eq)	$\frac{E \vdash M : T \quad E \vdash N : U \quad x \notin \text{fv}(M, N)}{E \vdash M = N : \{x : \text{bool} \mid (x = \text{true} \wedge M = N) \vee (x = \text{false} \wedge M \neq N)\}}$	
(Exp Assume)	(Exp Assert)	
$\frac{}{E \vdash \diamond} \quad \text{fnfv}(C) \subseteq \text{dom}(E)$	$\frac{}{E \vdash C}$	$\frac{}{E \vdash \text{assume } C : \{- : \text{unit} \mid C\}} \quad E \vdash \text{assert } C : \text{unit}$
(Exp Let)	$\frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin \text{fv}(U)}{E \vdash \text{let } x = A \text{ in } B : U}$	

##### Rules for Unit Type:

(Sub Unit)	(Val Unit)
$\frac{}{E \vdash \diamond}$	$\frac{}{E \vdash \diamond}$
$E \vdash \text{unit} <: \text{unit}$	$E \vdash () : \text{unit}$

##### Rules for Function Types:

(Sub Fun)	
$\frac{}{E \vdash T' <: T} \quad E, x : T' \vdash U <: U'$	$\frac{}{E \vdash (\Pi x : T. U) <: (\Pi x : T'. U')}$
(Val Fun)	
$\frac{}{E, x : T \vdash A : U}$	$E \vdash \text{fun } x \rightarrow A : (\Pi x : T. U)$

$$\frac{\text{(Exp Appl)} \quad E \vdash M : (\prod x : T. U) \quad E \vdash N : T}{E \vdash MN : U\{N/x\}}$$

### Rules for Pair Types:

$$\frac{\text{(Sub Pair)} \quad E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\sum x : T. U) <: (\sum x : T'. U')}$$

$$\frac{\text{(Val Pair)} \quad E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\sum x : T. U)}$$

$$\frac{\text{(Exp Split)} \quad E \vdash M : (\sum x : T. U) \quad E, x : T, y : U, - : \{(x, y) = M\} \vdash A : V \quad \{x, y\} \cap \text{fv}(V) = \emptyset}{E \vdash \text{let } (x, y) = M \text{ in } A : V}$$

### Rules for Sums and Recursive Types:

$$\frac{\text{(Sub Sum)} \quad E \vdash T <: T' \quad E \vdash U <: U' \quad \text{(Sub Var)} \quad E \vdash \diamond (\alpha <: \alpha') \in E}{E \vdash (T + T') <: (U + U') \quad E \vdash \alpha <: \alpha'}$$

$$\frac{\text{(Sub Rec)} \quad E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \text{fnfv}(T') \quad \alpha' \notin \text{fnfv}(T)}{E \vdash (\mu \alpha. T) <: (\mu \alpha'. T')}$$

$$\text{inl} : (T, T+U) \quad \text{inr} : (U, T+U) \quad \text{fold} : (T\{\mu \alpha. T/\alpha\}, \mu \alpha. T)$$

$$\frac{\text{(Val Inl Inr Fold)} \quad h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h M : U}$$

$$\frac{\text{(Exp Match Inl Inr Fold)} \quad E \vdash M : T \quad h : (H, T) \quad E, x : H, - : \{h x = M\} \vdash A : U \quad x \notin \text{fv}(U) \quad E, - : \{\forall x. h x \neq M\} \vdash B : U}{E \vdash \text{match } M \text{ with } h x \rightarrow A \text{ else } B : U}$$

### Rules for Refinement Types:

$$\frac{\text{(Sub Refine Left)} \quad E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \quad \frac{\text{(Sub Refine Right)} \quad E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

$$\frac{\text{(Val Refine)} \quad E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

### Rules for Concurrency:

$$\frac{\text{(Exp Res)} \quad E, a \uparrow T \vdash A : U \quad a \notin \text{fn}(U)}{E \vdash (va)A : U}$$

$$\frac{\text{(Exp Send)} \quad E \vdash M : T \quad (a \uparrow T) \in E}{E \vdash a!M : \text{unit}} \quad \frac{\text{(Exp Recv)} \quad E \vdash \diamond (a \uparrow T) \in E}{E \vdash a? : T}$$

$$\frac{\text{(Exp Fork)} \quad E, - : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, - : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \uparrow A_2) : T_2}$$

$$\frac{\overline{(va)A} = (\exists a. \overline{A}) \quad \overline{A_1 \uparrow A_2} = (\overline{A_1} \wedge \overline{A_2})}{\text{let } x = A_1 \text{ in } A_2 = \overline{A_1} \quad \text{assume } \overline{C} = C}$$

$\overline{A} = \text{True}$  if  $A$  matches no other rule

### 3.3 Properties of the Type System

Let  $E$  be *executable* if and only if  $\text{recvar}(E) = \emptyset$ .

**Lemma 2 (Static Safety)** *If  $\emptyset \vdash S : T$  then  $S$  is statically safe.*

**Proposition 3 ( $\Rightarrow$  Preserves Types)** *If  $E$  is executable and  $E \vdash A : T$  and  $A \Rightarrow A'$  then  $E \vdash A' : T$ .*

**Proposition 4 ( $\rightarrow$  Preserves Types)** *If  $E$  is executable,  $\text{fv}(A) = \emptyset$ , and  $E \vdash A : T$  and  $A \rightarrow A'$  then  $E \vdash A' : T$ .*

**Theorem 1 (Safety)** *If  $\emptyset \vdash A : T$  then  $A$  is safe.*

## 4 A Type System for Robust Safety

### Formal Threat Model: Opponents and Robust Safety

A closed expression  $O$  is an *opponent* iff  $O$  contains no occurrence of **assert**.

A closed expression  $A$  is *robustly safe* iff the application  $OA$  is safe for all opponents  $O$ .

To allow type-based reasoning about the opponent, we introduce a *universal type*  $\mathbf{Un}$  of data known to the opponent. Somewhat arbitrarily, we define  $\mathbf{Un} \triangleq \mathbf{unit}$ . By definition,  $\mathbf{Un}$  is type equivalent to (both a subtype and a supertype of) all of the following types:  $\mathbf{unit}$ ,  $(\Pi x : \mathbf{Un}. \mathbf{Un})$ ,  $(\Sigma x : \mathbf{Un}. \mathbf{Un})$ ,  $(\mathbf{Un} + \mathbf{Un})$ , and  $(\mu \alpha. \mathbf{Un})$ . Hence, we obtain *opponent typability*, that  $O : \mathbf{Un}$  for all opponents  $O$ .

It is useful to characterize two *kinds* of type: *public types* (of data that may flow to the opponent) and *tainted types* (of data that may flow from the opponent).

### Public and Tainted Types:

Let a type  $T$  be *public* if and only if  $T <: \mathbf{Un}$ .

Let a type  $T$  be *tainted* if and only if  $\mathbf{Un} <: T$ .

We add these kinds to our type system as follows.

### Syntax of Kinds:

$v ::= \mathbf{pub} \mid \mathbf{tnt}$  kind  
 Let  $\bar{v}$  satisfy  $\overline{\mathbf{pub}} = \mathbf{tnt}$  and  $\overline{\mathbf{tnt}} = \mathbf{pub}$ .

$\mu ::=$  environment entry  
 ... as before  
 $\alpha :: v$  kinding

$dom(\alpha :: v) = \{\alpha\}$

### Additional Judgment:

$E \vdash T :: v$  in  $E$ , type  $T$  has kind  $v$

### Kinding Rules: $E \vdash T :: v$ for $v \in \{\mathbf{pub}, \mathbf{tnt}\}$

(Kind Unit) (Kind Fun)  
 $\frac{E \vdash \diamond}{E \vdash \mathbf{unit} :: v} \quad \frac{E \vdash T :: \bar{v} \quad E, x : T \vdash U :: v}{E \vdash (\Pi x : T. U) :: v}$

(Kind Pair) (Kind Sum)  
 $\frac{E \vdash T :: v \quad E, x : T \vdash U :: v}{E \vdash (\Sigma x : T. U) :: v} \quad \frac{E \vdash T :: v \quad E \vdash U :: v}{E \vdash (T + U) :: v}$

(Kind Var) (Kind Rec)  
 $\frac{E \vdash \diamond \quad (\alpha :: v) \in E}{E \vdash \alpha :: v} \quad \frac{E, \alpha :: v \vdash T :: v}{E \vdash (\mu \alpha. T) :: v}$

(Kind Refine Public) (Kind Refine Tainted)  
 $\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}} \quad \frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$

### Additional Rule of Subtyping: $E \vdash T <: U$

(Sub Public Tainted)  
 $\frac{E \vdash T :: \mathbf{pub} \quad E \vdash U :: \mathbf{tnt}}{E \vdash T <: U}$

**Lemma 5 (Public Tainted)** For all  $T$  and executable  $E$ :

(1)  $E \vdash T :: \mathbf{pub}$  if and only if  $E \vdash T <: \mathbf{Un}$ .

(2)  $E \vdash T :: \mathbf{tnt}$  if and only if  $E \vdash \mathbf{Un} <: T$ .

**Lemma 6 (Opponent Typability)** Suppose  $E \vdash \diamond$  and that  $E$  is executable. If  $O$  is an expression containing no **assert** such that  $(a \downarrow \mathbf{Un}) \in E$  for each name  $a \in \text{fn}(O)$ , and  $(x : \mathbf{Un}) \in E$  for each variable  $x \in \text{fv}(O)$ , then  $E \vdash O : \mathbf{Un}$ .

**Theorem 2 (Robust Safety)** If  $\emptyset \vdash A : \mathbf{Un}$  then  $A$  is robustly safe.

## References

- J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement types for secure implementations. Technical Report MSR-TR-2008-118, Microsoft Research, 2008. A preliminary, abridged version appears in the proceedings of CSF'08.
- C. Gunter. *Semantics of programming languages*. MIT Press, 1992.
- A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3-4):289-360, 1993.