Principles and Applications of Refinement Types

Andrew D. Gordon and Cédric Fournet

October 2009

Technical Report
MSR–TR–2009–147

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

This tutorial article appears in the proceedings of the 2009 Marktoberdorf Summer School, *Logics and Languages for Reliability and Security*, editors Javiera Esparza, Bernd Spanfelner, and Orna Grumberg, NATO Science for Peace and Security Series D: Information and Communication Security - Volume 25, IOS Press 2010, pages 73–104.

The present version (of December 2010) corrects an error in Exercise 8.

Principles and Applications of Refinement Types

Andrew D. GORDON and Cédric FOURNET
Microsoft Research

Abstract. A refinement type $\{x : T \mid C\}$ is the subset of the type T consisting of the values x to satisfy the formula C. In this tutorial article we explain the principles of refinement types by developing from first principles a concurrent λ-calculus whose type system supports refinement types. Moreover, we describe a series of applications of our refined type theory and of related systems.

Keywords. Refinement types, FPC, RCF, F7

1. Introduction

A refinement type is a type qualified by a logical constraint; an example is the type of positive numbers, that is, the type $\{x : \mathsf{int} \mid x > 0\}$ of integers qualified by the is-greater-than-zero constraint. Although this idea has been known in the research community for some time, it has been assumed impractical because of the difficulties of constraint solving. But recent advances in automated reasoning have overturned this conventional wisdom, and transformed the idea into a practical design principle.

In these lecture notes, we develop from first principles a theory of refinement types for a concurrent λ-calculus. This theory is the foundation for a practical typechecker for refinement types. We describe the type system in detail and sketch a range of applications in language-based security, including verification and synthesis of security protocols and their implementations, and type-based analysis of web application security and code-based access control.

We begin in Section 2 with the syntax, operational semantics, and (unrefined) type system of the Fixpoint Calculus. This calculus, FPC for short, is a typed call-by-value λ-calculus with pairs, tagged unions, and iso-recursive types. It is a formal basis for pure functional programming in the core of languages like ML or Haskell. Section 3 extends FPC with concurrency and message-passing in the style of the π-calculus, so as to model imperative programming, concurrent functional programming, and also distributed communication protocols.

We introduce refinement types in Section 4, by adding them to the type system of Concurrent FPC. As well as developing soundness results for the type system (known as Refined Concurrent FPC, or RCF for short), we describe in detail some simple usages for refinements, for example, to specify communication protocols. To complement the theory, Section 5 outlines some substantial applications of refinement types, mostly based on F7 [4], an enhanced typechecker for F# [34] and Objective Caml [19]. Finally,

Section 6 surveys some of the historical development of refinement types and Section 7 concludes. We include exercises, of varying difficulty, and also proof sketches for some of the main theorems.

Exercise 1 *Complete all the proofs in the article. Hint: see the technical report [4].*

The web site `http://research.microsoft.com/en-us/people/adg/part.aspx` has additional material, including slides for Gordon's lectures on this topic at the 2009 Marktoberdorf Summer School.

2. FPC: Fixpoint Calculus

As a basis for subsequent sections on concurrency and on refinement types, we describe the core Fixpoint Calculus. This calculus first appears in lecture notes on domain theory and operational semantics by Plotkin [26], although the name FPC is introduced by Gunter [16].

We define the syntax, an operational semantics, and a type system. The syntax of the calculus is built up from expressions and values. Expressions denote computations. Values are the outcomes of computations. The operational semantics is a reduction relation $A \to A'$, meaning that the next step in the computation denoted by the expression A yields the expression A'. The main property we show of FPC is that the computation denoted by any closed well-typed expression either diverges or yields a unique value.

2.1. Syntax and Operational Semantics of FPC

We assume a countable set of variables, ranged over by x, y, and z. We also assume three value constructors, **inl**, **inr**, and **fold**, used to construct data.

The Fixpoint Calculus (FPC):

x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of iso-recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction
$A, B ::=$	expression
M	value
$M\ N$	application
let $(x, y) = M$ **in** A	pair split ($x \neq y$; scope of x, y is A)
match M **with** $h\ x \to A$ **else** B	constructor match (scope of x is A)
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)

We adopt some standard syntactic conventions concerning variable binding and substitution. The table indicates the scope of each bound variable. For example, in a function **fun** $x \to A$, the variable x is bound, with scope A. If an occurrence of a variable is not bound, we say that it is free. An expression is *closed* when it has no free variables. For any phrase of syntax ϕ (such as a value or expression or type), let $fv(\phi)$ be the set of variables occurring free in ϕ. We write $\phi\{\psi/x\}$ for the outcome of the capture-avoiding substitution of ψ for each free occurrence of x in the phrase ϕ. We identify phrases of syntax up to the consistent renaming of bound variables (also known as alpha-conversion).

We explain the syntax and intended semantics of values and then expressions. Every value is an expression, denoting itself. Apart from variables x, there are four kinds of value: unit, functions, pairs, and constructions.

- Unit, the empty-tuple, is written $()$.
- A function is written **fun** $x \to A$, where A is an expression.
- A pair is written (M, N), where M and N are values.
- A construction is written $h\ M$, where M is a value, and either $h \in \{\textbf{inl}, \textbf{inr}\}$ to represent tagged data, or $h = \textbf{fold}$ to represent data of recursive type.

Apart from values, there are the following kinds of expression:

- An application expression $M N$ applies the argument N to the value M, expected to be a function **fun** $x \to A$, resulting in the expression $A\{N/x\}$.
- A split-expression **let** $(x, y) = M$ **in** A decomposes the value M, expected to be a pair (N_1, N_2), resulting in the expression $A\{N_1/x_1\}\{N_2/x_2\}$.
- A match-expression **match** M **with** $h\ x \to A$ **else** B tests the value M, resulting in $A\{N/x\}$ when M is a construction $h\ N$, and in B otherwise.
- An equality-expression $M = N$ tests whether M and N are syntactically identical; if so, it yields **inr** $()$ (encoding true), and otherwise **inl** $()$ (encoding false).
- A let-expression **let** $x = A$ **in** B computes the sequential composition of A and B; it first evaluates A, yielding an outcome M, and then evaluates $B\{M/x\}$.

We formalize the intended semantics of expressions as a reduction relation, written $A \to A'$, which represents an individual step of the computation denoted by an expression. Reduction is the least relation on expressions closed under the following rules.

The Reduction Relation: $A \to A'$

$$(\textbf{fun}\ x \to A)\ N \to A\{N/x\} \qquad \text{(Red Fun)}$$

$$(\textbf{let}\ (x_1, x_2) = (N_1, N_2)\ \textbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\} \qquad \text{(Red Split)}$$

$$(\textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B) \to \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases} \qquad \text{(Red Match)}$$

$$M = N \to \begin{cases} \textbf{inr}() & \text{if } M = N \\ \textbf{inl}() & \text{otherwise} \end{cases} \qquad \text{(Red Eq)}$$

$$\textbf{let}\ x\ =\ M\ \textbf{in}\ A \to A\{M/x\} \qquad \text{(Red Let Val)}$$

$$A \to A' \Rightarrow \textbf{let}\ x\ =\ A\ \textbf{in}\ B \to \textbf{let}\ x\ =\ A'\ \textbf{in}\ B \qquad \text{(Red Let)}$$

The first five rules correspond to the five bullet points in our prose description of expressions. The final rule (Red Let) is a congruence rule that allows reductions within the first expression A in a let-expression **let** $x = A$ **in** B. We can apply (Red Let) to reduce the expression A until it reaches a value, and then (Red Let Val) applies.

We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$, so that $A \rightarrow^* A'$ means there is a reduction sequence $A = A_0 \rightarrow \cdots \rightarrow A_n = A'$ for $n \geq 0$.

A basic property of FPC is that reduction is deterministic:

Lemma 1 (Determinism) *If $A \rightarrow B$ and $A \rightarrow B'$ then $B = B'$.*

Proof: By induction on the structure of A. □

Another significant property is that reduction does not introduce new value or type variables. We are mainly concerned with the reductions of closed expressions, representing complete programs. The following lemma assures us that reductions from closed expression lead only to closed expressions.

Lemma 2 (Identifiers) *If $A \rightarrow A'$ then $fv(A') \subseteq fv(A)$.*

Proof: By induction on the derivation of $A \rightarrow A'$. □

2.2. Type System of FPC

The primary purpose of a type system is to prevent errors during the execution of expressions. The type system of FPC is based on assigning types to expressions to classify their possible values. We begin with the syntax of types.

Syntax of FPC Types:

$T, U, V ::=$	type
α	type variable
unit	unit type
$T \rightarrow U$	function type
$T \times U$	pair type
$T + U$	sum type
rec $\alpha.T$	iso-recursive type (scope of α is T)

The syntax of types is based on a countable set of type variables, ranged over by α, and disjoint from the set of value variables. We include the type variables as well as the value variables in the set $fv(\phi)$ of identifiers occurring free in a phrase of syntax ϕ. We write $T\{U/\alpha\}$ for the outcome of the capture-avoiding substitution of U for each free occurrence of α in the type T.

Apart from type variables, there are the following kinds of type:

- The unit type **unit** is the type of the unit value $()$.
- The function type $T \rightarrow U$ is the type of functions **fun** $x \rightarrow A$, that map values of type T to values of type U.
- The pair type $T \times U$ is the type of pairs (M, N) where M has type T and N has type U.

- The sum type $T + U$ is the type of constructions **inl**(M) where M has type T, and **inr**(N) where N has type U.
- The iso-recursive type **rec** $\alpha.T$ is the type of constructions **fold**(M) where M has type $T\{$**rec** $\alpha.T/\alpha\}$.

Our type system is formalized as *judgments*, formal sentences written $E \vdash \mathcal{J}$, where E is a *typing environment*, and $\mathcal{J}$ is a predicate. The typing environment E records information about all the identifiers in scope for the predicate $\mathcal{J}$. For FPC, each environment is a list $\mu_1, \ldots, \mu_n$ where each entry μ_i either declares a type variable, or a value variable together with its type. For each entry μ, the set $\mathit{dom}(\mu)$ consists of the identifiers defined by μ, while the set $\mathit{free}(\mu)$ consists of the identifiers used by μ. The set $\mathit{dom}(E)$ is the union of all the identifiers defined by entries in E.

Syntax of Static Typing Environments:

$\mu ::=$	environment entry
$\quad \alpha$	$\quad$ type variable
$\quad x : T$	$\quad$ variable typing
$E ::= \mu_1, \ldots, \mu_n$	environment (written $\varnothing$ when $n = 0$)

We write $\mu \in E$ to mean that μ is an entry in the list E.
$\mathit{dom}(\alpha) = \{\alpha\} \quad \mathit{free}(\alpha) = \varnothing$
$\mathit{dom}(x : T) = \{x\} \quad \mathit{free}(x : T) = \mathit{free}(T)$
$\mathit{dom}(\mu_1, \ldots, \mu_n) = \mathit{dom}(\mu_1) \cup \cdots \cup \mathit{dom}(\mu_n)$

The three judgments of the FPC type system are as follows.

Judgments of the Type System:

$E \vdash \diamond$	environment E is well-formed
$E \vdash T$	in environment E, type T is well-formed
$E \vdash A : T$	in environment E, expression A has type T

These three judgments are inductively defined by the rules in the following tables. Said otherwise, each judgment $E \vdash \mathcal{J}$ holds just if there is a proof tree whose nodes correspond to instances of the following rules.

Rules of Well-Formedness: $E \vdash \diamond \quad E \vdash T$

(Env Empty)	(Env Entry)	(Type)
	$E \vdash \diamond$	$E \vdash \diamond$
	$\mathit{free}(\mu) \subseteq \mathit{dom}(E) \quad \mathit{dom}(E) \cap \mathit{dom}(\mu) = \varnothing$	$\mathit{free}(T) \subseteq \mathit{dom}(E)$
$\varnothing \vdash \diamond$	$E, \mu \vdash \diamond$	$E \vdash T$

Exercise 2 *Write down the proof tree for $\alpha, \beta, x : \alpha \to \beta \vdash \diamond$. Notice that $y : \alpha \vdash \diamond$ is not derivable (because the variable α is not defined), but $\alpha, y : \alpha \vdash \diamond$ is derivable.*

Exercise 3 *Prove that in a well-formed environment, the identifiers used by each entry are defined, that is, if $E \vdash \diamond$ and $\mu \in E$ then $\mathit{free}(\mu) \subseteq \mathit{dom}(E)$.*

Exercise 4 *Prove that in a well-formed environment there is at most one entry per identifier, that is, if $E \vdash \diamond$ and $\mu_1 \in E$ and $\mu_2 \in E$ and $\mu_1 \neq \mu_2$ then $dom(\mu_1) \cap dom(\mu_2) = \varnothing$.*

Exercise 5 *Observe that if a type T is well-formed in E, then E itself is well-formed, that is, if $E \vdash T$ then $E \vdash \diamond$.*

Rules of Expression Typing: $E \vdash A : T$

(Val Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Val Unit)
$$\frac{E \vdash \diamond}{E \vdash () : \mathbf{unit}}$$

(Val Fun)
$$\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun}\ x \to A : (T \to U)}$$

(Exp Appl)
$$\frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

(Val Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M, N) : (T \times U)}$$

(Exp Split)
$$\frac{E \vdash M : (T \times U) \quad E, x : T, y : U \vdash A : V}{E \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ A : V}$$

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

$\mathbf{inl}{:}(T, T{+}U)$
$\mathbf{inr}{:}(U, T{+}U)$
$\mathbf{fold}{:}(T\{\mathbf{rec}\ \alpha.T/\alpha\}, \mathbf{rec}\ \alpha.T)$

(Exp Match Inl Inr Fold)
$$\frac{E \vdash M : T \quad h : (H, T) \quad E, x : H \vdash A : U \quad E \vdash B : U}{E \vdash \mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B : U}$$

(Exp Eq)
$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathbf{unit} + \mathbf{unit}}$$

(Exp Let)
$$\frac{E \vdash A : T \quad E, x : T \vdash B : U}{E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U}$$

The main purpose of these rules is to ensure that $E \vdash A : T$ implies that the value of expression A is of type T. When reading the rules, bear in mind an auxiliary property: that $E \vdash A : T$ is derivable only when E is well-formed. This auxiliary property is the reason why the rules (Val Var) and (Val Unit) require the environment to be well-formed. We rely on this property in the rules that deal with bound variables, namely (Val Fun), (Exp Split), (Exp Match Inl Inr Fold), and (Exp Let). In each of these rules we need to ensure that bound variables are distinct from the variables already in the typing environment (or else their types would be ambiguous). For example, in (Exp Let), we need to ensure that x does not occur already in E. Since the rule assumes the judgment $E, x : T \vdash B : U$, the auxiliary property implies that $E, x : T \vdash \diamond$, and from this that $x \notin dom(E)$ (see Exercise 4).

Exercise 6 *Prove that $E \vdash A : T$ implies $E \vdash T$. The auxiliary property follows.*

The following propositions and theorem establish soundness of the type system.

Proposition 1 (Preservation for FPC) *If $E \vdash A : T$ and $A \to A'$ then $E \vdash A' : T$.*

Proof: By induction on the derivation of the relation $A \to A'$. □

Proposition 2 (Progress for FPC) *If $\varnothing \vdash A : T$ then either A is a value, or there is A' with $A \to A'$.*

Proof: By induction on the derivation of $\varnothing \vdash A : T$. □

Theorem 1 *Every well-typed closed expression either diverges or yields a unique value.*

Proof: Consider any closed expression A such that $\varnothing \vdash A : T$ for some type T. If there is an unbounded computation $A \to A_1 \to A_2 \to \cdots$ we say that A diverges. Otherwise, there is a bounded computation $A \to A_1 \to A_2 \to \cdots \to A_n$ such that there is no A_{n+1} such that $A_n \to A_{n+1}$. By n applications of Proposition 1, we obtain that $\varnothing \vdash A_i : T$ for each $i \in 1..n$. We have that $\varnothing \vdash A_n : T$ and that there is no A_{n+1} such that $A_n \to A_{n+1}$. By Proposition 2, then, it must be that A_n is a value. Moreover, A_n is the unique value of A, since by Lemma 1 the reduction relation is deterministic. □

In contrast, expressions that are not well-typed may lead to execution errors. For example, the application $A = (M, N)\ L$ of the pair (M, N) to the value L cannot be reduced by the operational semantics, but neither does it diverge, nor is it a value. We say that such an expression is *stuck*, or that it has *gone wrong*. The purpose of our type system is to prevent such execution errors. The expression A is not well-typed because the only rule applicable to applications is (Exp Appl), and it would require that (M, N) have a function type, but the only rule applicable to pairs is (Val Pair), which would assign (M, N) a pair type, and pair types are not function types.

As in most type systems, typing is conservative in that there are some expressions whose execution cannot go wrong but that nonetheless are not well-typed. Consider the expression $A' = \textbf{match inl}\ ()\ \textbf{with inl}\ x \to x\ \textbf{else}\ A$ where A is the ill-typed expression above. This expression reduces to a value, that is, $A' \to ()$, but we cannot typecheck A' because the only applicable rule is (Exp Match Inl Inr Fold), which requires the else-clause A to have a type, even if A is unreachable, as in this example.

2.3. Deriving Programming Constructs within FPC

FPC is a parsimonious core calculus, kept small for the sake of a simple theory, but missing many features of actual programming languages. Still, we can directly encode most of the features of pure call-by-value functional programming, such as the fragment of ML absent side-effects, within FPC.

FPC has a reduced syntax (as in A-normal form [31]) where destructor-expressions, such as applications, splits, and constructor matches, act on values rather than arbitrary expressions. We can recover full applicative syntax by inserting suitable let-expressions, as follows. (We assume the inserted bound variables are fresh.)

Implicit Lets:

$(A, B) \triangleq \textbf{let}\ x = A\ \textbf{in let}\ y = B\ \textbf{in}\ (x, y)$
$h\ A \triangleq \textbf{let}\ x = A\ \textbf{in}\ h\ x$

$A\ B \triangleq \textbf{let}\ x = A\ \textbf{in let}\ y = B\ \textbf{in}\ x\ y$
$\textbf{let}\ (x, y) = A\ \textbf{in}\ B \triangleq \textbf{let}\ z = A\ \textbf{in let}\ (x, y) = z\ \textbf{in}\ B$
$\textbf{match}\ A\ \textbf{with}\ h\ x \to B\ \textbf{else}\ B' \triangleq \textbf{let}\ z = A\ \textbf{in match}\ z\ \textbf{with}\ h\ x \to B\ \textbf{else}\ B'$
$A = B \triangleq \textbf{let}\ x = A\ \textbf{in let}\ y = B\ \textbf{in}\ x = y$
$A; B \triangleq \textbf{let}\ x = A\ \textbf{in}\ B \qquad$ where x not free in B

To see that FPC includes divergent expressions and indeed is Turing complete, consider the recursive type $\Lambda = \textbf{rec}\ \alpha.(\alpha \to \alpha)$. This type admits the following simple embedding of the untyped λ-calculus within FPC.

Encoding the Untyped Call-By-Value λ-Calculus:

$\mathsf{app}(A, B) \triangleq \textbf{match}\ A\ \textbf{with fold}\ f \to f\ B\ \textbf{else fold}(\textbf{fun}\ x \to x)$
$[\![x]\!] \triangleq x \qquad [\![\lambda x.L]\!] \triangleq \textbf{fold fun}\ x \to [\![L]\!] \qquad [\![L_1\ L_2]\!] \triangleq \mathsf{app}([\![L_1]\!], [\![L_2]\!])$

The abbreviation $\mathsf{app}(A, B)$ evaluates A to a value $\textbf{fold}\ M$, and uses a match-expression to unfold the value by binding f to M. The else-branch is unreachable since every value of type Λ takes the form $\textbf{fold}\ M$, but nonetheless the syntax of match-expressions requires an else-branch, so we use the expression $\textbf{fold}(\textbf{fun}\ x \to x)$.

Exercise 7 *Suppose that $x_1, \ldots, x_n$ are the free variables of a term L of the λ-calculus. Prove that $x_1 : \Lambda, \ldots, x_n : \Lambda \vdash [\![L]\!] : \Lambda$.*

We may obtain a divergent expression in FPC by considering the standard λ-calculus combinator $\Omega = \Delta\ \Delta$ where $\Delta = \lambda x.x\ x$. We have that $[\![\Delta]\!] = \textbf{fold fun}\ x \to \mathsf{app}(x, x)$ and $[\![\Omega]\!] = \mathsf{app}([\![\Delta]\!], [\![\Delta]\!])$.

Exercise 8 *Check that $[\![\Omega]\!] \to A_1 \to \cdots \to A_n \to [\![\Omega]\!]$ for some $A_1, \ldots, A_n$. What is n?*

More generally, given any type T together with a value $V_T : T$, we construct a divergent expression $\Omega_T : T$ as follows:

Divergent Expression Ω_T at Type T:

$\Delta_T \triangleq \textbf{rec}\ \alpha.(\alpha \to T)$
$\delta_T : \Delta_T \to T \triangleq \textbf{fun}\ d \to \textbf{match}\ d\ \textbf{with fold}\ f \to f\ d\ \textbf{else}\ V_T$
$\Omega_T : T \triangleq \delta_T\ \textbf{fold}(\delta_T)$

As in the $\mathsf{app}(A, B)$ expression above, the else-branch in δ_T is unreachable, since d belongs to recursive type Δ_T; nonetheless, we depend on the existence of the value V_T of type T for the whole **match** expression to be typable according to the rule (Exp Match Inl Inr Fold). Hence, this construction does not work for a type such as $\textbf{rec}\ \alpha.\alpha$ that has no values.

For any recursive type $\textbf{rec}\ \alpha.T$ with a value $V_{T\{\textbf{rec}\ \alpha.T/\alpha\}}$ of the unfolded type $T\{\textbf{rec}\ \alpha.T/\alpha\}$, we define an $\mathsf{unfold}_{\textbf{rec}\ \alpha.T}$ operation, which is an inverse to **fold**.

Unfolding Values of an Iso-Recursive Type $\textbf{rec}\ \alpha.T$:

$\mathsf{unfold}_{\textbf{rec}\ \alpha.T}(M) \triangleq \textbf{match}\ M\ \textbf{with fold}\ x \to x\ \textbf{else}\ V_{T\{\textbf{rec}\ \alpha.T/\alpha\}}$

(An iso-recursive type is so called because the types $\mathbf{rec}\ \alpha.T$ and $T\{\mathbf{rec}\ \alpha.T/\alpha\}$ are not equivalent, but are isomorphic, as witnessed by the **fold** and $\mathsf{unfold}_{\mathbf{rec}\ \alpha.T}$ operations.)

Next, we define a type of Booleans, together with truth values and a conditional expression as follows. (We have already used this encoding in the rule (Red Eq) to represent the result of an equality test.)

Example: Booleans and Conditional Branching

$\mathsf{bool} \triangleq \mathbf{unit} + \mathbf{unit}$
$\mathbf{false} \triangleq \mathbf{inl}\ ()$
$\mathbf{true} \triangleq \mathbf{inr}\ ()$
$\mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ B' \triangleq \mathbf{match}\ A\ \mathbf{with}\ \mathbf{inr}\ \_ \to B\ \mathbf{else}\ B'$

Exercise 9 *Show that if* $\varnothing \vdash A : \mathsf{bool}$ *then either* A *diverges, or* $A \to^* \mathbf{true}$*, or* $A \to^* \mathbf{false}$*.*

Exercise 10 *Show that if* $A \to^* \mathbf{true}$ *then* $\mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ B' \to^* B$*, and similarly that if* $A \to^* \mathbf{false}$ *then* $\mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ B' \to^* B$*.*

Next, we define a type of natural numbers and arithmetic operations. Since there is a value $V_{\mathbf{unit}+\mathsf{nat}} = \mathbf{inl}\ ()$, we can rely on the derived unfolding operation $\mathsf{unfold}_{\mathsf{nat}}$.

Example: Arithmetic

$\mathsf{nat} \triangleq \mathbf{rec}\ \alpha.(\mathbf{unit} + \alpha)$
$\mathsf{zero} \triangleq \mathbf{fold}\ \mathbf{inl}\ ()$
$\mathsf{succ}(M) \triangleq \mathbf{fold}\ \mathbf{inr}\ M$
$\mathsf{iszero} \triangleq \mathbf{fun}\ x \to \mathbf{match}\ \mathsf{unfold}_{\mathsf{nat}}\ x\ \mathbf{with}\ \mathbf{inl}\ y \to \mathbf{true}\ \mathbf{else}\ \mathbf{false}$
$\mathsf{pred} \triangleq \mathbf{fun}\ x \to \mathbf{match}\ \mathsf{unfold}_{\mathsf{nat}}\ x\ \mathbf{with}\ \mathbf{inr}\ y \to y\ \mathbf{else}\ \mathsf{zero}$

Exercise 11 *Calculate the reductions:* $\mathsf{iszero}\ \mathsf{zero} \to^* \mathbf{true}$ *and* $\mathsf{iszero}\ (\mathsf{succ}(N)) \to^* \mathbf{false}$ *and* $\mathsf{pred}\ (\mathsf{succ}(N)) \to^* N$*.*

Exercise 12 *For any type* T*, consider the type* $(T)\mathsf{list} \triangleq \mathbf{rec}\ \alpha.(\mathbf{unit} + (T \times \alpha))$ *of lists of* T*. Derive list processing, that is, values* nil *and* $M :: N$*, and functions* null*,* hd*,* tl *such that* $\mathsf{null}\ \mathsf{nil} \to^* \mathbf{true}$ *and* $\mathsf{null}\ (M :: N) \to^* \mathbf{false}$ *and* $\mathsf{hd}\ (M :: N) \to^* M$ *and* $\mathsf{tl}\ (M :: N) \to^* N$*.*

Exercise 13 *For any function type* $T_f = T_1 \to T_2$*, define a fixpoint function* fix *with type* $(T_f \to T_f) \to T_f$ *such that* $\mathsf{fix}\ M\ N \to^* M\ (\mathsf{fix}\ M)\ N$*. State any additional assumption needed. (Hint: consider our construction of the divergent expression* Ω_T *.) Given such a function, we may encode ML-style recursive function definitions as follows:* $\mathbf{let\ rec}\ f x = A \triangleq \mathbf{let}\ f = \mathsf{fix}\ (\mathbf{fun}\ f\ x \to A)$*.*

Exercise 14 *Define a type* int *of (both positive and negative) integers, together with functions for equality, addition, subtraction, etc.*

Exercise 15 *We say that a type variable* α only occurs positively *in a type* T *if every free occurrence of* α *is to the left of an even number of function arrows in* T*. A recursive type* **rec** $\alpha.T$ *is* positive *if* α *only occurs positively in* T*. For example,* **rec** $\alpha.\beta \to \alpha$ *and* **rec** $\alpha.(\alpha \to \beta) \to \beta$ *are positive, because* α *occurs in* $\beta \to \alpha$ *and in* $(\alpha \to \beta) \to \beta$ *to the left of zero and two arrows, respectively. On the other hand,* **rec** $\alpha.\alpha \to \beta$ *is not positive, because the occurrence of* α *is to the left of one arrow.*

Show that every well-typed expression has a value in the fragment of FPC where each recursive type is positive. (Hint: show there is a translation of well-typed expressions into well-typed expressions of another calculus known to be normalizing (that is, where there are no unbounded reduction sequences). Consider the polymorphic λ*-calculus System F [14], or the variant of System F with positive types due to Mendler [21].)*

3. Concurrent FPC

In this section we extend FPC with concurrency and message-passing in the style of the π-calculus [22,32]. The resulting language, Concurrent FPC, can directly represent a wide range of language features, including mutable state, shared-memory parallel programming, dynamic allocation, and channel-based communication. Moreover, it can represent distributed security protocols, an important application area for the system of refinement types that we add to Concurrent FPC in the next section.

We obtain Concurrent FPC by augmenting the syntax, operational semantics, and type system of FPC with additional rules. The main result of the section is a preservation theorem, that the reduction relation preserve typing.

3.1. Syntax and Operational Semantics of Concurrent FPC

In FPC, an expression denotes a single thread of computation that either reduces to a unique value, or diverges. In contrast, when adding concurrency to FPC, an expression denotes a parallel collection of threads of computation. Each thread may reduce to a value, but along the way, as side-effects of reduction, it may send or receive messages on named channels, create new channels, and fork additional threads. Out of the collection of threads denoted by an expression, we distinguish a main thread, written by convention on the right, whose value is the value of the whole expression. A thread may diverge, as in FPC, but additionally a thread may deadlock, for example, if it blocks waiting for a message on a channel known only to itself.

We introduce a countable set of *names*, ranged over by a, b, and disjoint from the set of value and type variables. We include names as well as value and type variables in the set $fv(\phi)$ of identifiers occurring free in a phrase of syntax ϕ. We write $\phi\{a'/a\}$ for the operation of the capture-avoiding renaming of free occurrences of a in ϕ to a'. As before, a closed expression has no free variables, but it may have free names. Closed expressions represent run-time computations, in which all variables in source code have been bound to values, but there may be free names corresponding to fixed, global communication channels.

The syntax of values is unchanged but the syntax of expressions is extended as follows. (Names occur in expressions, and hence within function values **fun** $x \to A$, but names are not values in their own right.)

Additional Syntax:

a, b, c	names
$A, B ::=$	expression
$\ldots$	as in Section 2
$A \Rsh B$	fork (parallel composition)
$(\nu a)A$	restriction (name generation) (scope of a is A)
$a!M$	send M on channel a
$a?$	receive off channel a

- A parallel composition $A \Rsh B$ represents expressions A and B running in parallel. We consider B to be the main expression, and A to be an expression forked in the background.
- A restriction $(\nu a)A$ creates a fresh channel name a, then runs A.
- A send-expression $a!M$ returns $()$, and asynchronously produces message M on channel a.
- A receive-expression $a?$ blocks until it can consume some message M off channel a, and returns M.

In a composition $A \Rsh B$, the semantics is that there is a foreground expression B, whose value is the value of the whole composition, and a background expression A, whose value is ignored. In our syntax we arbitrarily place the main expression B on the right, and the forked expression A on the left. (In most process calculi, processes do not return values, and so the order of processes in a parallel composition does not matter.)

As a first example in Concurrent FPC, consider the following expression:

$$a!42 \Rsh (\nu c)((\textbf{let}\ x = a?\ \textbf{in}\ c!x) \Rsh (\textbf{let}\ y = c?\ \textbf{in}\ y))$$

When this executes, the leftmost thread $a!42$ sends 42 on channel a. The restriction (νc) creates a fresh channel c. The middle thread $\textbf{let}\ x = a?\ \textbf{in}\ c!x$ receives a message 42 off channel a, calls it x, then sends it on c. Finally, the rightmost thread, $\textbf{let}\ y = c?\ \textbf{in}\ y$, inputs 42 off c, calls it y, and then returns it as its value, and indeed as the value of the whole expression.

The operational semantics of Concurrent FPC consists of a reduction relation, written $A \to A'$, whose definition depends on an auxiliary *heating relation*, written $A \Rrightarrow A'$. The main purpose of heating $A \Rrightarrow A'$ is to re-arrange an expression A into a structurally similar form A' so that a reduction rule may be applied, but without changing the possible behaviour of A, including its value.

These two relations are defined by the rules from the previous section, together with the rules presented in the following four tables. These tables describe (1) core rules for the heating relation, (2) rules specific to parallel composition, (3) rules specific to restriction, and (4) rules specific to message-passing.

The Heating Relation: $A \Rrightarrow A'$

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

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)

$A \Rrightarrow A' \Rightarrow \textbf{let}\ x\ =\ A\ \textbf{in}\ B \Rrightarrow \textbf{let}\ x\ =\ A'\ \textbf{in}\ B$	(Heat Let)
$A \to A' \quad \text{if } A \Rrightarrow B, B \to B', B' \Rrightarrow A'$	(Red Heat)

(Heat Refl) and (Heat Trans) make heating reflexive and transitive, but heating is not symmetric in general. (Process calculi often use a symmetric version of heating, usually called *structural equivalence*.) (Heat Let) allows heating within let-expressions. (Red Heat) allows reductions enabled by first heating an expression.

Operational Semantics of Parallel Composition:

$() \Rsh A \equiv A$	(Heat Fork ())
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$	(Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$	(Heat Fork Comm)
$\textbf{let}\ x\ =\ (A \Rsh A')\ \textbf{in}\ B \equiv A \Rsh (\textbf{let}\ x\ =\ A'\ \textbf{in}\ B)$	(Heat Fork Let)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)
$A \to A' \Rightarrow (A \Rsh B) \to (A' \Rsh B)$	(Red Fork 1)
$B \to B' \Rightarrow (A \Rsh B) \to (A \Rsh B')$	(Red Fork 2)

The first group of rules above allows re-arrangements of parallel composition. (Heat Fork ()) asserts that a $()$ thread in the background makes no difference to an expression. The symmetric version $A \Rsh () \equiv A$ is not derivable and indeed unwanted, because this would change the foreground thread from $()$ to A, and hence may change the value of the expression. (Heat Fork Assoc) that the bracketing of parallel compositions does not matter. (Heat Fork Comm) asserts that the order of the background threads A and A' does not matter. Again, we do not want $(A \Rsh A') \Rrightarrow (A' \Rsh A)$ in general, because then heating would change the foreground thread; the value of $A \Rsh A'$ is the value of A', while the value of $A' \Rsh A$ is the value of A.

The second group above consists of congruence rules that allow heating and reduction to the left or right of composition.

Operational Semantics of Name Generation:

$a \notin \mathit{free}(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$	(Heat Res Fork 1)
$a \notin \mathit{free}(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$	(Heat Res Fork 2)
$a \notin \mathit{free}(B) \Rightarrow \textbf{let}\ x\ =\ (\nu a)A\ \textbf{in}\ B \Rrightarrow (\nu a)\textbf{let}\ x\ =\ A\ \textbf{in}\ B$	(Heat Res Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$	(Red Res)

A restriction $(\nu a)A$ represents the scope of a fresh channel name a. The rules (Heat Res Fork 1) and (Heat Res Fork 2) formalize the idea that these scopes are mobile, and in particular that they may expand to embrace an expression A' in parallel with a restriction $(\nu a)A$, unless the bound name a would be confused with existing occurrences of a in A'. We can always satisfy the side-condition that $a \notin \mathit{free}(A')$ by renaming the bound name. The rule (Heat Res Let) allows a restriction to move outside a let-expression.

The second group above consists of congruence rules that allow reduction and heating within restriction.

Operational Semantics of Message Passing:

$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$a!M \Rsh a? \to M$	(Red Comm)

The rule (Heat Msg ()) formalizes that message sending is asynchronous. A foreground output $a!M$ immediately returns $()$ and becomes a background thread. The reduction (Red Comm) represents the communication of a message M between background thread $a!M$ to a foreground thread $a?$, leaving the message M itself as the resulting foreground thread.

As we have discussed, parallel composition is not symmetric, and so we cannot directly apply (Red Comm) to allow the two threads in the expression $a? \Rsh a!M$ to communicate. Instead, we can apply the heating rules, including (Heat Msg ()), to deduce the following.

$$\begin{array}{rll} a? \Rsh a!M \Rrightarrow a? \Rsh (a!M \Rsh ()) & & \text{by (Heat Msg ()) and (Heat Fork 2)} \\ \Rrightarrow (a? \Rsh a!M) \Rsh () & & \text{by (Heat Fork Assoc)} \\ \Rrightarrow (a!M \Rsh a?) \Rsh () & & \text{by (Heat Fork Comm)} \\ \to M \Rsh () & & \text{by (Red Comm) and (Red Fork 1)} \end{array}$$

By combining the individual steps above with (Heat Trans), (Red Heat), and (Heat Refl), we obtain $a? \Rsh a!M \to M \Rsh ()$. This reduction step represents a message sent from the foreground thread to the background thread, resulting in the termination of both threads. The background thread terminates with the value M and the foreground terminates with the value $()$. (As usual, the value M of the background thread is ignored.)

Exercise 16 *Apply the heating and reduction rules to show that:*

$$a!42 \Rsh (\nu c)((\textbf{let}\ x = a?\ \textbf{in}\ c!x) \Rsh (\textbf{let}\ y = c?\ \textbf{in}\ y)) \to^* (\nu c)42$$

(Hint: use (Heat Res Fork 1) to pull (νc) to the top, use (Heat Fork Let) to float $a!42$ into the left-hand let-expression, apply (Red Comm) and (Red Let Val). The rest is similar.)

Exercise 17 *What are the reductions of the expression:* $a!3 \Rsh \textbf{let}\ x = a?\ \textbf{in}\ M\ x$

Exercise 18 *What are the reductions of the expression:* $a!\textbf{true} \Rsh a!\textbf{false}$

As the following exercises show, reduction is nondeterministic, and may deadlock.

Exercise 19 *What are the reductions of the expression:* $a!3 \Rsh a? \Rsh a!5$

Exercise 20 *One of the expressions* $(a?; b!()) \Rsh (b?; a!())$ *and* $(a!(); b?) \Rsh (b!(); a?)$ *is deadlocked. Which one?*

3.2. Type System of Concurrent FPC

We extend the type system of Section 2 to Concurrent FPC. The main new idea is that we extend the syntax of environment entries $a \updownarrow T$ to record the type T of messages allowed to be exchanged on the channel a.

Syntax of Typing Environments:

$\mu ::=$	environment entry
$\ldots$	as in Section 2
$a \updownarrow T$	channel typing

$dom(a \updownarrow T) = \{a\} \quad free(a \updownarrow T) = free(T)$

The three judgments have the same form as before ($E \vdash \diamond$, $E \vdash T$, and $E \vdash A : T$), and are defined by the rules in Section 2 together with the following rules.

Rules for Restriction, I/O, and Parallel Composition:

(Exp Fork)
$$\frac{E \vdash A_1 : T_1 \quad E \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}$$

(Exp Res)
$$\frac{E, a \updownarrow T \vdash A : U}{E \vdash (\nu a)A : U}$$

(Exp Send)
$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \textbf{unit}}$$

(Exp Recv)
$$\frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

The rule (Exp Fork) says that a composition $A_1 \Rsh A_2$ is well-typed when its components are, and that the type of the composition is the same as the type T_2 of the foreground expression A_2. The rule (Exp Res) says that a restriction $(\nu a)A$ is well-typed when its body is, assuming there is a type T for messages exchanged on the restricted name. The rule (Exp Send) says that a message send $a!M$ has type unit so long as the type of the message matches the type of the channel. The rule (Exp Recv) says that the result type of a message receive $a?$ is the type of the channel a.

The key soundness property of the resulting type system is that both heating and reduction preserve well-typing.

Lemma 3 *If $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.*

Proof: By induction on the derivation of the relation $A \Rrightarrow A'$. □

Proposition 3 (Preservation for Concurrent FPC) *If $E \vdash A : T$ and $A \to A'$ then $E \vdash A' : T$.*

Proof: By induction on the derivation of the relation $A \to A'$. □

The type system of Concurrent FPC rules out neither nondeterminism (see Exercise 19) nor deadlock (see Exercise 20). Hence, Concurrent FPC enjoys neither the progress property of FPC, Theorem 2, nor the property that every well-typed expression either diverges or reduces to a value

Exercise 21 *Find a well-typed expression that is not a value, but has no reductions.*

Still, as the following exercise shows, as in FPC, being well-typed does avoid execution errors arising from applying operations to the wrong sort of value.

Exercise 22 *Prove for every closed well-typed expression that (1) if it is an application $M\ N$ then the value M is a function; (2) if it is a split* **let** $(x, y) = M$ **in** A' *then the value M is a pair; (3) if it is a match* **match** M **with** $h\ x \to A'$ **else** B *then the value M is a construction.*

3.3. Deriving Programming Constructs within Concurrent FPC

Our core syntax ($a?$ and $a!M$ for message passing, $(\nu a)A$ for channel name creation, and $A \Rsh B$ for concurrency) is a mathematical notation in the style of process calculi. In applications it is convenient to represent these features as programming language functions, as in various extensions of ML with concurrency [18,20,28].

In the following representation of Concurrent ML primitives, a channel with name a is a pair $(!_a, ?_a)$ consisting of a send function $!_a = \textbf{fun}\, x \to a!x$ and a receive function $?_a = \textbf{fun}\ \_ \to a?$. (Recall that although names occur in the syntax of Concurrent FPC they are not themselves values.)

Example: Concurrent ML

$!_a \triangleq \textbf{fun}\, x \to a!x$	capability to send on a
$?_a \triangleq \textbf{fun}\ \_ \to a?$	capability to receive off a
$\mathsf{chan} \triangleq \textbf{fun}\ \_ \to (\nu a)(!_a, ?_a)$	create fresh channel
$\mathsf{send} \triangleq \textbf{fun}\, c\, x \to \textbf{let}\ (s, r) = c\ \textbf{in}\ s\ x$	send x on c
$\mathsf{recv} \triangleq \textbf{fun}\, c \to \textbf{let}\ (s, r) = c\ \textbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \textbf{fun}\, f \to (f() \Rsh ())$	run f in parallel

Exercise 23 *Define a type* $(T)\mathsf{chan} \triangleq (T \to \textbf{unit}) * (\textbf{unit} \to T)$*, where* $T \to \textbf{unit}$ *is the type of a send capability and* $\textbf{unit} \to T$ *is the type of a receive capability. Use this type to write down the types of the functions* chan*,* send*,* recv*, and* fork*.*

Hence, our running example can be written as the following ML code, assuming that variable a is bound to $(!_a, ?_a)$. The code is more verbose than the core syntax, but we can directly execute this code in ML.

```
fork (fun () → send a 42);
let c = chan() in
fork (fun () → let x = recv a in send c x);
let y = recv c in y
```

It is convenient to write systems models in this executable notation. Still, one may wonder if we have lost expressiveness by moving to Concurrent FPC instead of the π-calculus. We show below a basic reassurance, that there is a direct translation from π-calculus processes to the expressions of Concurrent FPC. A more sophisticated answer—beyond the scope of this article—would be to study the relationship between the formal semantics and behavioural equivalences of the two calculi.

We introduce the syntax and informal semantics of an asynchronous, polyadic π-calculus . Let u, v range over π-calculus values, each of which is either a variable x or a name a. An asynchronous polyadic output $u\langle v_1, \ldots, v_n\rangle$ represents the tu-

ple $\langle v_1, \ldots, v_n \rangle$ sent on channel u. A polyadic input $u(x_1, \ldots, x_n).P$ blocks until there is an output $u\langle v_1, \ldots, v_n \rangle$, and then may consume it, leading to the continuation process $P\{v_1/x_1\} \ldots \{v_n/x_n\}$. A composition $P \mid Q$ runs P and Q in parallel. (In the π-calculus, processes do not return results, and parallel composition is symmetric, that is, $P \mid Q$ behaves the same as $Q \mid P$.) A replication $!P$ acts like an unbounded array $P \mid P \mid P \mid \ldots$ of replicas of P running in parallel. A restriction $(\nu a)P$ creates a new channel a and runs P. A nil process $\mathbf{0}$ has no behaviour.

Encoding the Untyped Polyadic Asynchronous π-Calculus:

$[\![x]\!] \triangleq x$
$[\![a]\!] \triangleq (!_a, ?_a)$
$[\![u\langle v_1, \ldots, v_n \rangle]\!] \triangleq \mathsf{send}\ [\![u]\!]\ ([\![v_1]\!], \ldots, [\![v_n]\!])$
$[\![u(x_1, \ldots, x_n).P]\!] \triangleq \mathbf{let}\ x_1, \ldots, x_n = \mathsf{recv}\ [\![u]\!]\ \mathbf{in}\ [\![P]\!]$
$[\![P \mid Q]\!] = [\![P]\!] \Rsh [\![Q]\!] \Rsh ()$
$[\![!P]\!] = (\mathsf{fix}\ \mathbf{fun}\ f \to \mathbf{fun}\ x \to [\![P]\!] \Rsh f())\ ()$
$[\![(\nu a)P]\!] = (\nu a)[\![P]\!]$
$[\![\mathbf{0}]\!] = ()$

Exercise 24 *The encoding relies on n-ary tuple expressions and splitting, which are definable from the binary tuples that are primitive in FPC. For $n = 3$, we can define $(M_1, M_2, M_3) \triangleq (M_1, (M_2, M_3))$ and* $\mathbf{let}\ x_1, x_2, x_3 = M\ \mathbf{in}\ A \triangleq \mathbf{let}\ x_1, x_{23} = M\ \mathbf{in}\ \mathbf{let}\ x_2, x_3 = x_{23}\ \mathbf{in}\ A$ *for some fresh variable x_{23}. Define the encoding of n-ary tuples in general, for $n \geq 0$.*

Finally, we show how to represent heap-allocated mutable state using channels. In particular, we represent ML-style references, as follows.

Example: Mutable State

$(T)\mathbf{ref} \triangleq (T)\mathsf{chan}$

$\mathbf{ref}\ M \triangleq \mathbf{let}\ r = \mathsf{chan}()\ \mathbf{in}\ \mathsf{send}\ r\ M; r$	new reference to M
$!M \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ N$	update M with N

Exercise 25 *What are the reductions of the expression:* $\mathbf{let}\ x = \mathbf{ref}\ 5\ \mathbf{in}\ x := 7$

Exercise 26 *In his article in this volume, Xavier Leroy describes the prototypical imperative language IMP. Write a semantics for IMP in Concurrent FPC, based on storing each IMP variable in a ref.*

Exercise 27 *For those familiar with process calculi, define a labelled transition system for concurrent FPC, and prove its correspondence with the reduction semantics. Investigate behavioural equivalence for concurrent FPC.*

4. RCF: Refined Concurrent FPC

In this section, we obtain RCF, our final calculus, by extending expressions with logical assumptions and assertions, and by extending types with logical refinements to ensure that all assertions follow from prior assumptions. Operationally, the resulting language is essentially unchanged: assumptions and assertions do not affect evaluation; they are inserted only to specify the expected properties of the computation. On the other hand, refinement types are much more expressive, inasmuch as types now carry logical formulas and typechecking now involves logical deductions. The section ends with detailed typing examples.

4.1. Syntax and Operational Semantics of RCF

RCF is a calculus parameterized by a logic, which is used to specify the properties of its computations. We introduce a new syntactic category of logical formulas, ranged over by C. We use the values of RCF (M) as logical terms, and we build formulas from predicates on terms, of the form $p(M_1, \ldots, M_n)$. These predicates includes at least equality, written $M = N$, interpreted as syntactic equality of values up to alpha-conversion. In the following, we use the standard syntax and semantics of first-order logic, with conjunctions, disjunctions, existential quantifiers, and so on. We write for instance $C \wedge C'$ and $C \Rightarrow C'$ for logical conjunction and implication, respectively.

As illustrated below, the choice of a particular logic depends on the target verification properties, and also on the availability of provers for the logic. Hence, to reason about integer arithmetic and array bounds, we may let C range over conjunctions of equalities and inequations between integers. Formally, our semantics and typing theorems for RCF apply to any logic that meets a series of basic properties, given in detail elsewhere [4]. For instance, we require that deducibility be closed by extension (adding hypotheses C_i) and by value substitution.

A General Class of Logics:

$C ::= p(M_1, \ldots, M_n) \mid M = M' \mid \ldots$	
$\{C_1, \ldots, C_n\} \vdash C$	deducibility relation

We now extend our operational semantics to formalize the notion of a global set of formulas, the *log*, drawn from some logic. The log collects all assumptions that have been made during a particular run. RCF values and expressions are those of Concurrent FPC, supplemented with the following:

Additional Syntax:

$A, B ::=$	expression
$\ldots$	as in Section 3
assume C	logical assumption
assert C	logical assertion

Formulas appear in the syntax of values and expressions only as the parameters of assumptions and assertions.

- When executed, an assumption introduces a logical formula, deemed to be true, and records it as part of a global knowledge on the computation. For example, the code that sends a request with content 42, may include the expression **assume** Request(42) to record this fact.
- When executed, an assertion claims that a given logical formula should logically follow from prior assumptions. When this is not the case, the computation does not follow its logical specification. The main purpose of refinement types is to statically exclude such specification errors.

Additional Rules of Heating and Reduction:

assume $C \Rrightarrow$ **assume** $C \Rsh ()$	(Heat Assume ())
assert $C \to ()$	(Red Assert)

Operationally, these expressions immediately yield $()$. To evaluate **assume** C, add C to the log, and return $()$. To evaluate **assert** C, return $()$. If C logically follows from the logged formulas, we say the assertion *succeeds*; otherwise, we say the assertion *fails*.

To specify expression safety, we apply the heating relation to rewrite expressions in normal form, up to renaming and reordering of auxiliary threads. These normal forms are named *structures*, and ranged over by $\mathbf{S}$.

Structures and Static Safety:

$e ::= M \mid MN \mid M = N \mid$ **let** $(x, y) = M$ **in** $B \mid$
match M **with** $h\ x \to A$ **else** $B \mid M? \mid$ **assert** C
$\prod_{i \in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \ldots \Rsh A_n$
$\mathcal{L} ::= \{\} \mid (\textbf{let}\ x = \mathcal{L}\ \textbf{in}\ B)$

$$\mathbf{S} ::= (\nu a_1) \ldots (\nu a_\ell) \left(\left(\prod_{i \in 1..m} \textbf{assume}\ C_i \right) \Rsh \left(\prod_{j \in 1..n} c_j!M_j \right) \Rsh \left(\prod_{k \in 1..o} \mathcal{L}_k\{e_k\} \right) \right)$$

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

In a structure, all name restrictions are lifted to the top level (to ensure that all threads are in the same scope) and all active threads are flattened, then grouped depending on their elementary expression: either an assumed formula—after applying (Heat Assume ())—or a pending message on a channel—after applying (Heat Msg ())—or some other elementary expression e_k in a let-context $\mathcal{L}$. (These let-contexts represent the continuations to be executed when the expressions e_k complete.) Hence, structures represent the global state of the computation, with a log of assumed formulas, a store of pending messages, and a run-queue of expressions being evaluated in parallel contexts.

In a given structure, some of the active expressions e_k may be assertions, of the form **assert** C'_k. We define static safety by requiring that, for each such assertion, the asserted formula C'_k logically follow from the current content of the log, seen as a conjunction of logical formulas $C_1, \ldots, C_m$.

The following lemma show that every expression can be heated into a structure. This enables us to lift our definition of safety from structures to expressions.

Lemma 4 *For every expression A, there is a structure $\mathbf{S}$ such that $A \Rrightarrow \mathbf{S}$.*

Proof: By induction on A and definition of the heating relation. □

Exercise 28 *Show that, for a given expression, static safety does not depend on the choice of a particular structure. Which properties of the logic does this rely on?*

Exercise 29 *Let A be an expression that uses concurrency only to implement ML-style references, as defined at the end of Section 3.3. Suppose that $A \rightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$. What is the shape of the structure $\mathbf{S}$?*

We are now ready to define our semantic notion of safety for expressions at run-time: an expression is *safe* when, after any sequence of reductions, any executable assertion follows from the current log, as specified on structures.

Expression Safety:

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

Exercise 30 *Which of the following expressions are safe when the formula C is $x = 1$? When C is $x > 0$? When C is $x = 0 \wedge x > 0$?*

(1) **let** $x = c?$ **in assert** C
(2) $a!1 \Rsh$ **let** $x = a?$ **in assert** C
(3) $a!1 \Rsh$ **let** $x = a!2 \Rsh a?$ **in assert** C
(4) $a!1 \Rsh$ **let** $x = a?$ **in** $a!2 \Rsh$ **assert** C

4.2. Type System of RCF

We now extend our type system to ensure that any well-typed expression is safe.

The types for RCF are similar to but more precise than those of FPC. As explained in the introduction, they include *refinement types*, of the form $\{x : T \mid C\}$. An expression A of type T can be given this more precise type when, for any value M it may reduce to at run-time, the formula $C\{M/x\}$ holds in its evaluation context. Said otherwise, this type ensures that the expression **let** $x = A$ **in** (**assert** $C; x$) is safe.

In general, A may be a sub-expression within the scope of other variables (for instance, A may be the body of a function with formal parameter y), and the refinement formula C may have free variables that refer to any value in scope (for instance x and y).

Thus, our types are value-dependent, since they include formulas that include values. This leads to the following extended syntax of types for RCF, with dependent function- and pair-type constructors in addition to refinement types.

Syntax of RCF Types:

$H, T, U, V ::=$	type	
	unit	unit type
	$\{x : T \mid C\}$	refinement type (scope of x is C)
	$\Pi x : T.\ U$	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
rec $\alpha.T$	iso-recursive type (scope of α is T)
α	iso-recursive type variable

- A value of $\{x : T \mid C\}$ is a value M of type T such that $C\{M/x\}$ holds
- A value of type $\Pi x : T.\ U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$. This subsumes function types $T \to U$ (in case x does not occur in U).
- A value of $\Sigma x : T.\ U$ is a pair (M, N) such that M has type T and N has type $U\{M/x\}$. This subsumes function types $T \times U$ (in case x does not occur in U).

Hence, functions can now be given types of the form $\Pi y : \{y : T \mid C'\}.\ \{x : U \mid C\}$ with two refinements, C' on its formal argument y, and C on y and its result x. Intuitively, this types specifies that the function has precondition C' and postcondition C. Anticipating on the typing rules, we may thus type the function **fun** $y \to y + 2$ as $\Pi y : \{y : \mathsf{int} \mid y \geq 0\}.\ \{x : \mathsf{int} \mid x > y\}$.

We extend environments with entries that record recursive variables during subtyping, and we let $\mathit{recvar}(E)$ be just the type variables occurring in subtyping entries of E.

Syntax of Typing Environments:

$\mu ::=$	environment entry
$\ldots$	as in Section 3
$\alpha <: \alpha'$	subtype ($\alpha \neq \alpha'$)

$\mathit{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\} \quad \mathit{free}(\alpha <: \alpha') = \varnothing$
$\mathit{recvar}(E) = \{\alpha, \alpha' \mid (\alpha <: \alpha') \in E\}$

The RCF type system involves five judgments, as follows:

Judgments of the Type System:

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

These five judgments are inductively defined by rules that are either given below or are identical to those presented in Sections 2 and 3. For instance, well-formedness is still defined by the rules (Env Empty), (Env Entry), and (Type) of Section 2, applied to our extended definitions for types and type environments.

We first explain how to extract and prove formulas from a typing environment, and then we define the subtyping relation and typing for expressions.

An important idea of refinement typing is to treat the typing environment as a conservative approximation of the logical formulas that will hold at run-time. For instance, if a function receives an argument with refined type $\{y : T \mid C\}$, then its body can be typechecked under the assumption that formula C will hold for y at run-time. To this

end, we let $\mathsf{forms}(E)$ collect the logical refinements for all entries in the typing environment E, and we rely on a new auxiliary judgment, $E \vdash C$, stating that a formula is deducible from the formulas of a given environment.

Rules for Formula Derivation:

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup \mathsf{forms}(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

(Derive)

$$\frac{E \vdash \diamond \quad \mathit{free}(C) \subseteq \mathit{dom}(E) \quad \mathsf{forms}(E) \vdash C}{E \vdash C}$$

In the rule (Derive), the first two hypotheses ensure that E is well-formed and defines any variable that occurs free in C. The third hypothesis refers to logical deducibility. In implementations of typechecking, this proof obligation is usually passed to an auxiliary theorem prover. For instance, typechecking the integer function above will involve proving that $y + 2 > y$.

Exercise 31 *A handy abbreviation is* $\{C\} \triangleq \{\_ : \mathbf{unit} \mid C\}$*, where* _ *stands for a fresh variable. We refer to such types as ok-types. What is* $\mathsf{forms}(x : \{C\})$*?*

Exercise 32 *What is* $\mathsf{forms}(x_1 : \{y_1 : \mathsf{int} \mid \mathsf{Even}(y_1)\}, x_2 : \{y_2 : \mathsf{int} \mid \mathsf{Odd}(x_1)\})$*?*

Instead of a simple type int for all integers, refinement types enables us to give many different types to the same value. For instance, 1 may be given any of the types int, $\{x : \mathsf{int} \mid x = 1\}$, $\{x : \mathsf{int} \mid x > 0\}$, $\{x : \mathsf{int} \mid x \neq 42\}$, $\{x : \mathsf{int} \mid x > 0 \wedge x \neq 42\}$, or even, assuming that y is a variable in scope, the type $\{x : \mathsf{int} \mid y > 0 \Rightarrow x = 1\}$.

Accordingly, expression typing relies on an auxiliary subtyping judgment, written $E \vdash T <: U$, which lets us convert between different logical variants of refinement types in a given context. Informally, a "smaller" refinement type is a type whose formulas are more precise: if $A : T$ and $T <: U$ then $A : U$.

We begin with subtyping and typing rules for refinements.

Rules for Refinement Types:

(Sub Refine Left)

$$\frac{E \vdash T <: T' \quad E \vdash \{x : T \mid C\}}{E \vdash \{x : T \mid C\} <: T'}$$

(Sub Refine Right)

$$\frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

(Val Refine)

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

For subtyping, (Sub Refine Left) states that a refinement is always more precise than its base type—the second hypothesis only requires that the refinement be well-formed. In contrast, (Sub Refine Right) states that a refinement may be added to form a supertype of T only when its formula C logically follows from the formulas in environment E. For typing, the rule (Val Refine) states that any value may be refined with any valid formula.

Exercise 33 *How would we derive* $\vdash \{x : \mathsf{int} \mid x > 0\} <: \mathsf{int}$.

Exercise 34 *Derive the following subtyping rules:*

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

Next, we give the typing rules for assumptions and assertions, which relate refinements to our semantic notion of expression safety.

Rules for Assume and Assert:

$$\frac{E \vdash \diamond \quad \mathit{free}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{assume}\ C : \{\_ : \mathbf{unit} \mid C\}}\ (\text{Exp Assume}) \qquad \frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathbf{unit}}\ (\text{Exp Assert})$$

The rule (Exp Assume) states that an assumption is always typable, as long as its formula is well-formed, and records the assumed formula in an ok-type. Hence, typechecking within the scope of the assume can use formula C to prove other formulas. Since typechecking accepts any formula, assumptions in programs should be carefully reviewed. The rule (Exp Assert) checks that C is deducible from the formulas in the current typing environment. Otherwise, typechecking fails.

The rules for functions and pairs are dependent variants of those given in Section 2. As regards subtyping, functional arguments are contravariant (with subtyping hypothesis $T' <: T$ rather than $T <: T'$), whereas functional results and pair projections are covariant.

Rules for Function Types:

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

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

Rules for Pair Types:

$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')}\ (\text{Sub Pair}) \qquad \frac{E \vdash M : T \quad E \vdash N : U\{M/x\}}{E \vdash (M, N) : (\Sigma x : T.\ U)}\ (\text{Val Pair})$$

$$\frac{E \vdash M : (\Sigma x : T.\ U) \quad E, x : T, y : U, \_ : \{(x, y) = M\} \vdash A : V \quad \{x, y\} \cap \mathit{fv}(V) = \varnothing}{E \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ A : V}\ (\text{Exp Split})$$

Exercise 35 *Understand why* $\vdash (\Pi x : \mathsf{int}.\ \mathsf{bool}) <: (\Pi x : \{x : \mathsf{int} \mid x > 0\}.\ \mathsf{bool})$ *but not* $\vdash (\Pi x : \{x : \mathsf{int} \mid x > 0\}.\ \mathsf{bool}) <: (\Pi x : \mathsf{int}.\ \mathsf{bool})$.

We also refine our typing rule for equality, to record the outcome of the dynamic equality test as a refinement on its Boolean result.

Rule for Equality Test:

(Exp Eq)

$$\frac{E \vdash M : T \qquad E \vdash N : U \qquad x \notin \mathit{fv}(M, N)}{E \vdash M = N : \{x : \mathsf{bool} \mid (x = \mathbf{true} \wedge M = N) \vee (x = \mathbf{false} \wedge M \neq N)\}}$$

We include from previous sections the typing rules (Exp Let), (Val Var), (Val Unit), (Val Inl Inr Fold), (Exp Match Inl Inr Fold), (Exp Res), (Exp Send), and (Exp Recv) unchanged, except that the rules (Exp Let), (Exp Match Inl Inr Fold), and (Exp Res) for expressions with bound identifiers need additional side-conditions to ensure that the bound identifiers do not appear in the type of the whole expressions. For example, the rule (Exp Res) for assigning a type U to a restriction $(\nu a)A$ needs the side-condition that $a \notin \mathit{free}(U)$.

We give below some standard rules for subtyping.

Standard Rules of (Dependent) Subtyping:

(Exp Subsum)

$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

(Sub Refl)

$$\frac{E \vdash T \qquad \mathit{recvar}(E) \cap \mathit{free}(T) = \varnothing}{E \vdash T <: T}$$

(Sub Var)

$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(Sub Sum)

$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

(Sub Rec)

$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathit{free}(T') \quad \alpha' \notin \mathit{free}(T)}{E \vdash (\mathbf{rec}\ \alpha.T) <: (\mathbf{rec}\ \alpha'.T')}$$

Exercise 36 *Prove that* $E \vdash T <: T'$ *is decidable, assuming an oracle for* $E \vdash C$.

Exercise 37 *(Hard.) Prove that* $E \vdash T <: T'$ *is transitive.*

Finally, here is a revised version of the rule for typing threads in parallel.

Dependent Rule for Fork:

(Exp Fork)

$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \qquad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}$$

$$\overline{(\nu a)A} = (\exists a.\overline{A})$$
$$\overline{A_1 \Rsh A_2} = (\overline{A_1} \wedge \overline{A_2})$$
$$\overline{\mathbf{let}\ x = A_1\ \mathbf{in}\ A_2} = \overline{A_1}$$
$$\overline{\mathbf{assume}\ C} = C$$
$$\overline{A} = \mathsf{True} \quad \text{if } A \text{ matches no other rule}$$

This rule relies on an auxiliary function from expressions A to formulas $\overline{A}$ that inductively collects all the active assumptions of A and returns their conjunction. (The

first case, for name restriction, approximates a local name as an existential value.) Hence, the assumptions of A_1 are available for typing A_2, and the converse.

Exercise 38 *Let* $\mathbf{S}$ *be a structure, as defined in Section 4.1. Suppose that* $\varnothing \vdash \mathbf{S}$. *Show that the active expressions of* $\mathbf{S}$ *can be typed in an environment that includes an ok-type for each of the assumed formulas* $C_1, \ldots C_m$.

We can now state our main theorem for RCF, which guarantees that any well-typed closed expression is indeed safe. (In the theorem statement, the typing environment accounts for the free names of A.)

Theorem 2 (Safety) *If* $a_1 \updownarrow T_1, \ldots, a_n \updownarrow T_n \vdash A : T$ *then* A *is safe.*

Exercise 39 *Which of the expressions of Exercise 30 are typable? Can you provide other examples of expressions that are safe but not typable?*

Exercise 40 *Relying on the refinement provided by rule (Exp Eq), establish a derived typing rule for the conditional expression* **if** $M = N$ **then** A **else** B.

Can we prove the judgment $x : \mathsf{int} \vdash$ **if** $x = 0$ **then** (**if** $x = 1$ **then assert** $(0 = 1)$)*?*

Exercise 41 *To confirm that RCF is a refinement of Concurrent FPC, let us recursively define refinement erasure on types, by* $(\{x : T \mid C\})^\# = T^\#$, *and on expressions, by* (**assume** $C)^\# = ()$ *and* (**assert** $C)^\# = ()$. *(1) Show that* $\varnothing \vdash A : T$ *in RCF implies* $\varnothing \vdash A^\# : T^\#$ *in Concurrent FPC. (2) Show that* $\varnothing \vdash A^\# : T^\#$ *in Concurrent FPC implies* $\{\mathsf{False}\} \vdash A : T$ *in RCF (where* False *is a contradiction, that is, a formula that entails any other formula).*

4.3. Examples of Typing Expressions in RCF

Typing the Running Example. In the following version of our running example from Section 3, the assumption **assume** $\mathsf{Sent}(x)$ marks the intention of the middle thread to send the message x, while the assertion **assert** $\mathsf{Sent}(x)$ marks the expectation of the rightmost thread that the received message has been sent from the middle thread.

$$a!42 \Rsh (\nu c)((\textbf{let } x = a? \textbf{ in assume } \mathsf{Sent}(x) \Rsh c!x) \Rsh (\textbf{let } y = c? \textbf{ in assert } \mathsf{Sent}(y)))$$

To type this, the key idea is that while the initial message on a has type int, the subsequent message on channel c has the refined type $\{x : \mathsf{int} \mid \mathsf{Sent}(x)\}$. We calculate the typing derivation for the three threads as follows, where we define $A_2 = (\textbf{assume } \mathsf{Sent}(x) \Rsh c!x)$ and $A_3 = \textbf{assert } \mathsf{Sent}(y)$.

(1) $a \updownarrow \mathsf{int} \vdash a!42 : \textbf{unit}$
by (Exp Send).

(2) $a \updownarrow \mathsf{int}, c \updownarrow \{x : \mathsf{int} \mid \mathsf{Sent}(x)\}, x : \mathsf{int} \vdash A_2 : \textbf{unit}$
by (Exp Fork), (Exp Assume), (Exp Send), noting $\overline{\textbf{assume } \mathsf{Sent}(x)} = \mathsf{Sent}(x)$.

(3) $a \updownarrow \mathsf{int}, c \updownarrow \{x : \mathsf{int} \mid \mathsf{Sent}(x)\}, y : \{x : \mathsf{int} \mid \mathsf{Sent}(x)\} \vdash A_3 : \textbf{unit}$
by (Exp Assert).

(4) $a \updownarrow \mathsf{int} \vdash (\nu c)(\textbf{let } x = a? \textbf{ in } A_2 \Rsh \textbf{let } y = c? \textbf{ in } A_3) : \textbf{unit}$
by (2), (3), (Exp Fork), and (Exp Res).

(5) $a \updownarrow \mathsf{int} \vdash a!42 \Rsh (\nu c)(\textbf{let } x = a? \textbf{ in } A_2 \Rsh \textbf{let } y = c? \textbf{ in } A_3) : \textbf{unit}$
by (1), (4), and (Exp Fork).

Hence, by Theorem 2, it follows that the example is safe.

Typing a Request/Response Protocol. As a second example, the code below implements a simple request/response protocol. A call service s f creates a replicated service that repeatedly consumes request messages (x, r) from the channel s, computes the value y of f x, and sends y on the reply channel r. A call client s x creates a client to invoke the service on channel s with request x.

```
let rec service (s:service) (f:int→ int) : unit =
  let x,r = (recv s):service_payload in
  assert(Request(x));
  let y = f x in
  assume (Response(x,y));
  send r y;
  service s f
```

```
let client (s:service) (x:int) =
  let r = chan() in
  assume (Request(x));
  send s (x,r);
  let y = recv r in
  assert(Response(x,y));
  y
```

The client code logs Request(x) to mark its intent to request service, while the server code logs Response(x, y) to mark that y is its reply to the request x. The service's assertion of Request(x) indicates its expectation that it has received a legitimate request, while the client's assertion of Response(x, y) indicates its expectation that the answer y that it has just received is indeed a response to the data x sent in its previous request.

We can typecheck this code using the following types (in a notation like that implemented by the F7 typechecker).

$$(;x)\mathsf{reply} \triangleq \{y : \mathsf{int} \mid \mathsf{Response}(x, y)\}$$

$$\mathsf{service} \triangleq (\Sigma x : \{x : \mathsf{int} \mid \mathsf{Request}(x)\}.\ (;x)\mathsf{reply})\mathsf{chan}$$

A value of type service is a channel for pairs (x, r) where x is an int such that Request(x) holds, and r is a channel of type $(;x)$reply, that is, a channel for sending integers y such that Response(x, y) holds. Even though the original request x is not returned on the reply channel r with the response y, the type of r mentions x, and hence can guarantee to the recipient that Response(x, y) holds. Again, we can apply Theorem 2 to guarantee the safety of systems modelled with the functions service and client.

The two examples in this section show the basic techniques of modelling distributed protocols via parallel processes and message passing on channels, of specifying safety properties using assumptions and assertions, and of verifying these properties using refinement types in RCF. These tutorial examples are small and rather abstract, but as we outline in the following section, the underlying techniques scale up to modelling and verifying substantial amounts of code.

Exercise 42 *Develop a theory of session types for RCF. Hmm, maybe this needs RSM, or some concurrent version of it.*

5. Some Applications of RCF

We describe a series of applications of RCF, where refinement types are used to establish a series of properties, mainly concerning security.

5.1. F7: An Implementation of RCF

F7 [6] is a refinement-typechecker for F# closely related to RCF [4]; its distribution includes a series of programming examples and libraries.

F7 supports modular typechecking. It takes as input both refinement-typed module interfaces (e.g. `part.fs7`) and F# module implementations (e.g. `part.fs`). When typechecking succeeds, it also generates plain F# module interfaces (e.g. `part.fsi`) obtained by erasing all type refinements. Since refinements play no role at run-time, F7 can be used for verifying code developed and compiled using F# tools and libraries. For instance, it is usually a good idea to typecheck first using F#, then using F7.

The F7 typechecking algorithm implements the typing rules of RCF, with some specific extensions: it performs partial type inference, implements type- and value-polymorphism, carefully controls the application of subtyping, and directly supports algebraic datatypes, whereas RCF formally relies on their encoding into sums. F7 uses first-order logic for its refinements, and Z3 for logical verification. Each time it applies a typing rule with a (non-trivial) logical deducibility condition, F7 passes a query to Z3.

5.2. Verifying Cryptographic Protocols and their Implementations

The main application of F7 so far is the security verification of cryptographic protocols and their implementations. In contrast with earlier type systems and verification tools dedicated to cryptography, most of the verification effort consists in developing reusable, typed libraries for a range of cryptographic primitives and protocol patterns—see [4] and [7] for their detailed description.

Digital Signatures In these notes, we only outline the use of refinement types for digital signatures. The purpose of a digital signature is to guarantee the authenticity of a piece of data, naturally modelled as a logical refinement. For simplicity, let us assume that signatures are used only to authenticate requests, of type $\{x : \mathsf{int} \mid \mathsf{Request}(x)\}$. (A more realistic refinement for security would also include the originator of the request, its header, and its intended recipient.)

Using a trusted channel, as illustrated in Section 4, one can directly exchange integers with that type. However, this implicitly assumes that all programs with access to the channel are well-typed in F7. In contrast, integers exchanged on a public, untrusted channel can only be given the plain type int, inasmuch as anyone may inject any integer value. To *dynamically* verify that a received integer is a genuine request, one needs to receive a signature and to cryptographically verify that signature using a trusted key. For this example, the cryptographic primitives for signing a message, and for verifying a message signature may be given the following types.

$$\begin{array}{l} \mathit{sign} \quad : \mathsf{skey} \to \{x : \mathsf{int} \mid \mathsf{Request}(x)\} \to \mathsf{bytes} \\ \mathit{verify} : \mathsf{vkey} \to \Pi x : \mathsf{int}.\ (\mathsf{bytes} \to \{b : \mathsf{bool} \mid b = \mathbf{true} \Rightarrow \mathsf{Request}(x)\})) \end{array}$$

where skey is the type of signing keys, vkey is the type of verification keys, and bytes is the type of signatures (an array of bytes). The refinement formula Request(x) can be seen as a contract between signers and verifiers. It appears as a precondition for calling the sign function, and as a postcondition after successfully calling the verify function. (When verify returns **false**, nothing is learned about x.) For more complex protocols, this refinement formula would cover all the potential usages for signing and verification keys.

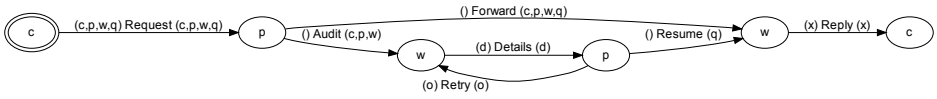
In this example, values of type int and bytes can be freely exchanged on public channels, but this is not the case for key types. For instance, if we send a signing key on an untrusted channel, then anyone can sign any value, whether or not it is a genuine request. As explained in [4] and [7] respectively, one can flexibly control which values can be exchanged on a public network and when, using either *kinds* or a particular form of *refinements*, and show that typechecking then suffices to ensure security against any active network attacker.

CardSpace F7 has been applied to the verification of authorization, authenticity and secrecy properties for a variety of protocol implementations. As an example, the CardSpace protocol for federated identity management builds upon standards for XML security, leading to complex message formats with a dozen cryptographic operations per message. We have typechecked a functional reference implementation of this protocol, initially developed for verification using a cryptographic protocol analyzer [8], and confirmed that modular verification by typing yields the same guarantees but scales much better than global proof techniques [7].

Zero-Knowledge Proofs Backes and others are independently developing F5, another implementation of RCF for cryptographic verification [2]. Their type system extends RCF with features including union and intersection types so as to support non-interactive zero-knowledge proofs of knowledge. In terms of refinements, this yields formulas existentially-quantified over the values used to build the cryptographic proof. The "zero-knowledge" aspect of the model is reflected by the fact that, from the viewpoint of the proof verifier, these values occur only in the refinements, much like ghost variables, and are thus not available at run-time. Their main case study is the verification of authenticity properties for the Direct Anonymous Attestation Protocol embedded in TPMs.

5.3. Security Protocol Synthesis for Multiparty sessions [5]

Many distributed protocols can be specified as a fixed, global graph where the nodes represent protocol participants and the edges represent their exchanged messages. This sort of specification simplifies the task of each protocol participant, who knows in advance which messages will be exchanged, and in what order. For instance, the picture below represents a session with three participants (a client, a proxy, and a web server); each edges carries a message label plus variables that indicate the message payloads.



However, on an open network with untrusted participants, there is usually no way to guarantee that remote participants will actually follow this multiparty session discipline.

Instead, each participant must carefully check whether each messages it receives is valid at each point of the session.

This is the purpose of a prototype multiparty-session compiler that automatically generates, implements, and verifies a communication protocol with enough cryptographic checks to ensure that, from the viewpoint of the session participants that use this defensive protocol, every message complies with the global session discipline. This session compiler is fairly complex, but it need not be trusted. Instead, to establish the correctness of the generated protocol code, the compiler also generates detailed refinement types that embed the structure of the global session graph and the causal dependencies between all messages. (For a 3-party session with loops, the generated code is a few thousand lines of F#, and the generated annotation is a few thousand lines of F7 refinement-type declarations.) The F7 typechecker is then called to verify that the protocol code indeed enforces the global session discipline.

For example, the generated refinement-type for the final Reply message from the web server to the client indicates that the proxy must have received a Request message with some query q from the client

5.4. Security of Multi-Tier Web Programming [3]

A *multi-tier programming language* is a high level web programming language that runs as code split between the multiple tiers of a web application. For example, Links [11] is a multi-tier programming language based on functional programming; a single Links source file results in application code to run in a web server, to HTML and JavaScript to run in the browser, and to SQL to run in the database.

An ideal of high level languages is that review of source code, and its source-level semantics, should suffice to establish security properties of compiled code [1]. In fact, a common problem with multi-tier languages is that there are low-level attacks on the compiled code that are not apparent from inspection of the source code. For example, the Links compilation strategy involve storing some session state, such as variable bindings, within HTML forms in the browser. This strategy makes Links vulnerable to an untrustworthy user reading or modifying these variable bindings. In particular, like some other web applications, Links is vulnerable to price-changing attacks, where if a shopping cart is held as variable bindings in the browser, the customer can modify the cart before checking out so as to obtain items at reduced prices.

To address this problem, an improved compilation strategy for Links is to rely on keyed hashes and encryption to protect the secrecy and integrity of web application data held in HTML forms. To validate the strategy, a type and effect system for Links source code is developed to allow reasoning about integrity properties, such as tracking that prices held in a shopping cart have indeed been obtained from the price database. Using the models of cryptography described in Section 5.2, the secure compilation strategy can be modelled in RCF, to show that the source-based properties proved with the type and effect system are preserved by the implementation.

5.5. Auditability by Typing [17]

Security protocols often rely on a pre-established mechanism to resolve conflicts: for a given *auditable property* on the outcome of the protocol, all protocol participants agree

on a fixed resolution procedure, the *judge*, and they log data obtained during the protocol run, the *evidence*. After the protocol completes, and if there is a conflict, every participant that rightfully followed the protocol should have collected enough evidence to convince the judge of their claim. This security property is called *auditability*.

For a target auditable property C with free variable x, say of type int, the judge function has an F# type of the form int $\rightarrow$ evidence $\rightarrow$ bool. The judge is *correct* when it returns **true** only when formula C holds for its first parameter substituted for x. An *audit point* in a protocol is a place where a participant collects evidence E that C holds for some value M substituted for x. This evidence is *complete* when the judge will always return **true** when presented with value M and collected evidence E.

Both correctness and completeness can be verified by refinement typing, as follows: find (and typecheck) a logical refinement D for the pair (x, E) at every audit point; and check that the judge function can be given the refinement type:

$$\Pi x : \mathsf{int}.\ \Pi e : \mathsf{evidence}.\ \{b : \mathsf{bool} \mid (b = \mathbf{true} \Rightarrow C) \wedge (D \Rightarrow b = \mathbf{true})\}$$

The first postcondition is for correctness; the second postcondition is for completeness of the evidence. For instance, the evidence may be a digital signature, the judge may cryptographically verify that this is a valid signature on x using some authorized key (with verification postcondition C), and the complete-evidence property D may state that the digital signature has been verified at every audit point.

5.6. Verifying Stateful Properties with the Refined State Monad [9]

A classical idea in pure functional programming is to represent imperative code as a function of the type shown below, known as the *state monad*.

$$\Pi s_0 : \mathsf{state}.\ \Sigma x : T.\ \mathsf{state}$$

A function of this type is a state-passing imperative computation. The function accepts an initial state $s_0 : \mathsf{state}$, and returns a result type $x : T$, paired with a final state. For example, the function $\mathsf{get} \triangleq \mathbf{fun}\, s_0 \rightarrow (s_0, s_0)$ is the computation to fetch and return the current state. Another example is $\mathsf{set}(M) \triangleq \mathbf{fun}\ s_0 \rightarrow ((), M)$, the computation to replace the current state with a new state $M : \mathsf{state}$.

The *refined state monad* is the refined function type $\{(s_0)C_0\}\, x : T\, \{(s_1)C_1\}$ obtained by adding refinement formulas C_0 and C_1 to the initial and final states.

$$\{(s_0)C_0\}\, x : T\, \{(s_1)C_1\} \triangleq \Pi s_0 : \{s_0 : \mathsf{state} \mid C_0\}.\ \Sigma x : T.\ \{s_1 : \mathsf{state} \mid C_1\}$$

In particular, we have the following types for the get and $\mathsf{set}(M)$ computations.

$$\mathsf{get} : \{(s_0)\mathsf{True}\}\, x : \mathsf{state}\, \{(s_1)(s_1 = s_0) \wedge (x = s_0)\}$$
$$\mathsf{set}(M) : \{(s_0)\mathsf{True}\}\, x : \mathbf{unit}\, \{(s_1)(s_1 = M)\}$$

A paper [9] derives a calculus based on the refined state monad in RCF, and investigates its application to checking code for compliance with state-based access control mechanisms. State-based access control deals with the situation when both trusted and untrusted code share the same execution environment. We need to allow trusted code to

access privileged resources, such as the file system, and to prevent untrusted code from doing so. With state-based access control, the runtime system maintains state representing the set of permissions currently available to each thread of control. Calls to sensitive library functions, such as to access the file system, only succeed if sufficient permissions are available, and otherwise throw security exceptions. As execution proceeds, the current permissions are modified both implicitly by function calls and returns, and explicitly by system calls to add and subtract permissions.

In practice, state-based access control is hard to program. A common problem arises from unnecessary security exceptions when trusted code is running with insufficient permissions, due to implicit updates to the state. Pottier, Skalka, and Smith [27] pioneered a state-sensitive type system to help detect such unnecessary security exceptions early. The idea is that being well-typed implies that expressions raise no security exceptions, so that code that may lead to exceptions is flagged statically by the typechecker.

We sketch how the system of Pottier, Skalka, and Smith can be recast and extended using the refined state monad. We choose state to be sets of permissions. For example, $\mathsf{ReadFile}(M) : \mathsf{state}$ is the set containing just the right to read the file M. Permissions form a lattice. We use the predicate $\mathsf{Subsumed}(M, N)$ to mean that the permissions M are less than the permissions N, and assume the axioms for a lattice. Hence, for example, the library function $\mathsf{readFile}(M)$ has the following type:

$$\mathsf{readFile}(M) : \{(s_0)\mathsf{Subsumed}(\mathsf{ReadFile}(M), s_0)\}\, x : \mathsf{string}\, \{(s_1)(s_1 = s_0)\}$$

The type says that $\mathsf{readFile}(M)$ can only be called when the current permissions s_0 include at least $\mathsf{ReadFile}(M)$, and that on return the current permissions are unchanged. See [9] for further explanations of the refined state monad and its applications.

6. Further Reading

Operational Semantics The textbook by Gunter [16] describes the operational semantics of FPC, together with a domain-theoretic denotational semantics and type system. Gunter's book also explains the basic techniques of syntax up to alpha-conversion, structural induction, and inductive definitions.

Type Systems For more background on type systems see Cardelli's article [10] or Pierce's book [25]. The division of type safety into progress and preservation theorems is due to Wright and Felleisen [35].

Process Calculi There are two standard textbooks on the π-calculus by Milner [22] and Sangiorgi and Walker [32]. Pierce and Sangiorgi [24] introduced typed channels with subtyping for the π-calculus. (Their system allows the send and receive capabilities for a channel to be communicated separately, and to have separate types). A reduction semantics with an asymmetric fork operation $A \Rsh B$ appears first in a concurrent object calculus [15].

Refinement Types Refinement types are also known as *subset types* (in constructive type theory [23]) or *predicate subtypes* (in the PVS prover [30]). More recently, a mechanism of subset types has been added to Coq [33]. The refinement types of Pfenning and Freeman [13] are subsets of preexisting inductive definitions; for instance, the even numbers

are a refinement of the natural numbers. Subsequently, DML [36] allows a more general form of refinement types, where base types may be refined by Boolean expressions. The expressions that may appear in DML refinements are known as *indexes*, and are restricted so as to make typechecking decidable. Hybrid typechecking [12] is the later idea that if static heuristics fail to settle whether the refinement expression in a refinement type is satisfied, instead of rejecting the program we can instead insert code to check the refinement expression at run time.

A cost of using refinement types in DML and other systems is that the programmer must provide many type annotations including refinement formulas. Liquid types (Logically Qualified Data Types) [29] are a specialized form of refinement type suitable for type inference. A liquid type takes the form $\{x : T \mid Q_1 \wedge \cdots \wedge Q_n\}$ where each qualifier Q_i is an instance of a finite user-supplied collection of predicates. Given an ML expression and the corresponding ML type derivation, liquid type inference generates a liquid type derivation by generating a formula variable for each unrefined type in its input, and then it applies an iterative algorithm to find a finite conjunction of qualifiers to assign to each formula variable. The algorithm shows promise; it can automatically infer refinement types for a variety of DML benchmarks (but ported to Objective Caml), with far fewer user-supplied types or formulas than in the original DML files.

7. Conclusions

We explained the principles of refinement types in terms of the concurrent λ-calculus RCF. We presented some applications in terms of a series of projects based on F7, an enhanced typechecker for F# and Objective Caml, whose theoretical foundation is RCF. We outlined the historical development of refinement types and described some other approaches.

We have found that several pre-existing type systems can be implemented as refinement types within RCF and F7 by assuming suitable predicates and axioms. It seems to us likely that many more existing and new static analyses may be discovered to be instances of refinement types, and can usefully and efficiently be implemented within this general framework. Consider these tutorial notes an invitation to learn the foundations of this framework, to discover its deficiencies, and to stretch it further.

Acknowledgements The basis for these tutorial notes is the calculus RCF together with its typechecker F7. We have cited various studies making use of F7; none of these would have been possible without the work of our esteemed colleague Karthik Bhargavan. He is the main developer of F7, and our collaborator on the original research on RCF and F7, along with Jesper Bengtson and Sergio Maffeis, to whom we are also grateful. Thanks also to Misha Aizatulin, François Dupressoir, Cătălin Hriţcu, and Pat Rondon, who commented on a draft of these notes.

References

[1] M. Abadi. Protection in programming-language translations. In *Automata, Languages and Programming: 25th International Colloquium (ICALP'98)*, volume 1443, pages 868–883. Springer, 1998.

[2] M. Backes, C. Hriţcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs - work in progress. In *Workshop on Foundations of Computer Security*, 2009.

[3] I. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2009)*, pages 27–38, 2009.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. 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.

[5] K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 124–140. IEEE Computer Society, July 2009.

[6] K. Bhargavan, C. Fournet, and A. Gordon. F7: Refinement types for F#. Available at http://research.microsoft.com/en-us/projects/F7/, 2008. Version 1.0.

[7] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456. ACM, 2010.

[8] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135. ACM Press, 2008.

[9] J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. Technical Report MSR–TR–2009–97, Microsoft Research, 2009.

[10] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS. Springer-Verlag, 2006.

[12] C. Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.

[13] T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*, pages 268–277. ACM, 1991.

[14] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[15] A. D. Gordon and P. Hankin. A concurrent object calculus: reduction and typing. In *3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, volume 16 of *ENTCS*. Elsevier, 1998.

[16] C. Gunter. *Semantics of programming languages*. MIT Press, 1992.

[17] N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability by typing. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 168–183. Springer, 2009.

[18] S. Holmström. PFL: A functional language for parallel programming. In *Declarative Programming Workshop*, pages 114–139. University College, London, 1983. Extended version published as Report 7, Programming Methodology Group, Chalmers University. September 1983.

[19] INRIA. *The Caml Language*. At `http://caml.inria.fr`.

[20] D. C. J. Matthews. Papers on Poly/ML. Technical Report 161, University of Cambridge Computer Laboratory, Feb. 1989.

[21] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991. Earlier version in LICS'88.

[22] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[23] B. Nordström and K. Petersson. Types and specifications. In *IFIP'83*, 1983.

[24] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[25] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[26] G. D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes, CSLI, Stanford University, July 1985.

[27] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM TOPLAS*, 27(2):344–382, 2005.

[28] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 294–305, June 1991.

[29] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*, pages 159–169. ACM, 2008.

[30] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[31] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic*

Computation, 6(3–4):289–360, 1993.

[32] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. CUP, 2001.

[33] M. Sozeau. Subset coercions in Coq. In *TYPES'06*, volume 4502 of *LNCS*, page 237. Springer, 2007.

[34] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

[35] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[36] H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM, 1999.