Pict Language Definition

Version 4.1

Benjamin C. Pierce

Computer Science Department
Indiana University
Lindley Hall 215
Bloomington
Indiana 47405-4101
USA
`pierce@cs.indiana.edu`

David N. Turner

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
United Kingdom
`dnt@dcs.gla.ac.uk`

March 22, 1998

This draft definition of the Pict language is still being checked for consistency and completeness. Bug reports appreciated!

Contents

Copying

Chapter 1

Introduction

Milner, Parrow, and Walker's π-calculus [EN86, MPW92, Mil91] generalizes the channel-based communication of CCS and its relatives [Mil80, Mil89, etc.] by allowing channels to be passed as data along other channels. This extension introduces an element of *mobility*, enabling the specification and verification of concurrent systems with dynamically evolving communication topologies. Channel mobility leads to a surprising increase in expressive power, yielding a calculus capable of describing a wide variety of high-level concurrent features [San92, San94a, San94b, San93, BS94, Mil90, Jon93, Wal95, Ama94, AP94, Wal94, etc.] while retaining a simple semantics and tractable algebraic theory.

A similar combination of simplicity and expressiveness has made the λ-calculus both a popular object of theoretical investigation and an attractive basis for sequential programming language design. By analogy, then, one may wonder what kind of high-level programming language can be constructed from the π-calculus.

$$\frac{\text{ML, Haskell, } \ldots}{\lambda\text{-calculus}} = \frac{?}{\pi\text{-calculus}}$$

A number of programming language designs have combined π-calculus-like communication with a functional core language [Rep91, BMT92, Mat91, GMP89, Hol83, Car86, Ber93, etc.], but none have gone so far as to take communication as the sole mechanism of computation.

The primary motivation of the Pict project, begun at the University of Edinburgh in 1992, was to design and implement a high-level concurrent language purely in terms of the π-calculus primitives. This report describes the syntax and formal sematics of the Pict language.

This document is not intended as a tutorial on Pict programming, and makes no attempt to explain or motivate the Pict language design. Interested readers are directed to the Pict tuturial [Pie96], the Pict compiler documentation [PT97a], the Pict standard libraries manual [PT97c], and a high-level survey of the Pict language design [PT97b].

Pict is an experimental language, and we welcome any comments or suggestions as to how we might improve the language itself or our presentation of its formal semantics. The purpose of writing down a formal language definition is not to standardise Pict, but to provide a formal framework which can be used as a basis for formulating and testing new language features. This document will therefore change as we continue to develop the language (the version number given on the title page indicates which version of the Pict compiler this document refers to).

1.1 Outline

After some notational preliminaries (Chapter 2), Chapter 3 describes the rules for lexical analysis and presents a grammar for the surface syntax of Pict programs. Chapter 4 discusses some features (such as separate compilation and inline C code) that are provided by the compiler but not considered part of the formal language definition. Chapter 5 shows how some very simple syntactic conveniences are desugared during parsing. Chapter 6 defines how variable scopes are resolved.

Chapters 7 and 8 define the kinding and subtyping relations. Chapter 9 then gives a system of typing rules for fully annotated programs. Chapter 10 gives the rules for type reconstruction, the process by which programs with some omitted type annotations are transformed into fully annotated programs.

Chapter 11 gives a few additional derived forms of similar complexity to the ones listed in Chapter 5 but requiring the complete type annotations added during type reconstruction to do their work. Chapter 12 then presents one final set of derived forms. These—the only non-local derived forms—are responsible for performing the CPS-conversion that transforms Pict programs with complex values such as nested applications into a pure message-passing core language very similar to the π-calculus.

The operational semantics of the core language is given in Chapter 13. We treat issues of fairness and details of the connection between Pict programs and the outside world informally.

Chapter 2

Preliminaries

We first introduce some notation and remark on some general conventions used in the rest of the document.

2.1 Metavariable Conventions

We use different metavariable names to denote different syntactic categories, as follows:

`a`	abstraction
`c`, `x`, `y`	variable
`d`	declaration
`e`	process expression
`FT`	field type
`fv`	field value
`fp`	field pattern
i, j, k, m, n	natural numbers
`k`	constant
`K`	kind
`l`	field label (either `x=` or else empty)
`p`	pattern
`path`	path
`S`, `T`, `U`	type
`RT`	reconstructable type (either `T` or else empty)
`v`	value
`X`, `Y`	type variable
Γ, Δ	typing context

2.2 Metasyntactic Conventions

In the grammar in Chapter 3, the possible forms of each production are listed on successive lines. Keywords are set in typewriter font. An expression of the form $X \ldots X$ denotes a list of zero or more occurrences of X. The expression $\langle empty \rangle$ denotes an empty production.

Chapter 3

Syntax

This chapter describes the syntax of Pict programs.

3.1 Lexical Rules

Whitespace characters are space, newline, tab, and formfeed (control-L). Comments are bracketed by `{-` and `-}` and may be nested. A comment is equivalent to whitespace.

Integers are sequences of digits (negative integers start with a `-` character). Strings can be any sequence of characters and escape sequences enclosed in double-quotes. The escape sequences `\"`, `\n`, and `\\` stand for the characters double-quote, newline, and backslash. The escape sequence `\ddd` (where `d` denotes a decimal digit) denotes the character with code `ddd` (codes outside the range `0..255` are illegal). Character constants consist of a single quote character (`'`), a character or escape sequence, and another single quote.

Alphanumeric identifiers begin with a symbol from the following set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Subsequent symbols may contain the following characters in addition to those mentioned above:

```
0 1 2 3 4 5 6 7 8 9 '
```

Symbolic identifiers are non-empty sequences of symbols drawn from the following set:

```
~ * % \ + - < > = & | @ $ , `
```

3.2 Reserved Words

The following symbols are reserved words:

`and`	`Bool`	`ccode`	`Char`	`def`	`else`	`false`	`if`	`import`	`inline`
`Int`	`new`	`now`	`rec`	`run`	`String`	`then`	`Top`	`true`	`type`
`Type`	`val`	`where`	`with`	`@`	`^`	`\`	`/`	`.`	`;`
`:`	`=`	`|`	`!`	`#`	`?`	`_`	`<`	`>`	`->`
`{`	`(`	`[`	`}`	`)`	`]`				

(The curly brackets `{` and `}` are not currently used, but are reserved for future expansion.)

3.3 Concrete Syntax

For each syntactic form, we note whether it is part of the core language (C), a derived form (D), an optional type annotation that is filled in during type reconstruction if omitted by the prorammer (R), or an extra-linguistic feature (E).

3.3.1 Compilation units

TopLevel	=	*Import* ... *Import Dec* ... *Dec*	*E*	Compilation unit
Import	=	`import` *String*	*E*	Import statement

3.3.2 Declarations

Dec	=	`new` *Id* `:` *Type*	*C*	Channel creation
		`val` *Pat* `=` *Val*	*D*	Value binding
		`run` *Proc*	*D*	Parallel process
		Val `;`	*D*	Sequential execution
		`inline def` *Id Abs*	*D*	Inlinable definition
		`def` Id_1 Abs_1 `and ... and` Id_n Abs_n	*C*	Recursive definition ($n \geq 1$)
		`type` *Id* `=` *Type*	*D*	Type abbreviation
		`type (` *Id* $KindedPolarityId_1$ ... $KindedPolarityId_n$ `) =` *Type*	*D*	Type operator abbrev ($n \geq 1$)
		`now (` *Id Flag* ... *Flag* `)`	*E*	Compiler directive
Flag	=	*Id*	*E*	Ordinary flag
		Int	*E*	Numeric flag
		String	*E*	String flag

3.3.3 Abstractions

Abs	=	*Pat* `=` *Proc*	*C*	Process abstraction
		`(` *Label FieldPat* ... *Label FieldPat* `)` *RType* `=` *Val*	*D*	Value abstraction

3.3.4 Patterns

Pat	=	*Id RType*	*C*	Variable pattern
		`[` *Label FieldPat* ... *Label FieldPat* `]`	*C*	Record pattern
		`( rec` *RType Pat* `)`	*C*	Rectype pattern
		`_` *RType*	*C*	Wildcard pattern
		Id RType `@` *Pat*	*C*	Layered pattern
FieldPat	=	*Pat*	*C*	Value field
		`#` *Id Constr*	*C*	Type field

3.3.5 Type constraints

Constr	=	⟨*empty*⟩	*D*	No constraint
		`<` *Type*	*C*	Subtype constraint
		`:` *Kind*	*C*	Kinding constraint
		`=` *Type*	*C*	Equality constraint

3.3.6 Processes

Proc	=	*Val* `!` *Val*	*C*	Output atom
		Val `?` *Abs*	*C*	Input prefix
		`( )`	*C*	Null process
		`(` $Proc_1$ `| ... |` $Proc_n$ `)`	*C*	Parallel composition ($n \geq 2$)
		`(` Dec_1 ... Dec_n *Proc* `)`	*C*	Local declarations ($n \geq 1$)
		`if` *Val* `then` *Proc* `else` *Proc*	*C*	Conditional

3.3.7 Values

Val	=	*Const*	C	Constant
		Path	C	Path
		`\` *Abs*	D	Process abstraction
		`[` *Label FieldVal* … *Label FieldVal* `]`	C	Record
		`if` *RType Val* `then` *Val* `else` *Val*	D	Conditional
		`(` *Val RType* `with` *Label FieldVal* … *Label FieldVal* `)`	D	Field extension
		`(` *Val RType* `where` *Label FieldVal* … *Label FieldVal* `)`	D	Field override
		`(` *RType Val Label FieldVal* … *Label FieldVal* `)`	D	Application
		`(` *Val* `>` $Val_1 \ldots Val_n$ `)`	D	Right-assoc application ($n \geq 2$)
		`(` *Val* `<` $Val_1 \ldots Val_n$ `)`	D	Left-assoc application ($n \geq 2$)
		`(` `rec` *RType Val* `)`	C	Rectype value
		`(` $Dec_1 \ldots Dec_n$ *Val* `)`	D	Local declarations ($n \geq 1$)
		`(` `ccode` *Int Id String FieldVal* … *FieldVal* `)`	E	Inline C code
Path	=	*Id*	C	Variable
		Path `.` *Id*	C	Record field projection
FieldVal	=	*Val*	C	Value field
		`#` *Type*	C	Type field
Const	=	*String*	C	String constant
		Char	C	Character constant
		Int	C	Integer constant
		`true`	C	Boolean constant
		`false`	C	Boolean constant

3.3.8 Types

Type	=	`Top`	C	Top type
		Id	C	Type identifier
		`^` *Type*	C	Input/output channel
		`!` *Type*	C	Output channel
		`/` *Type*	C	Responsive output channel
		`?` *Type*	C	Input channel
		`Int`	C	Integer type
		`Char`	C	Character type
		`Bool`	C	Boolean type
		`String`	C	String type
		`[` *Label FieldType* … *Label FieldType* `]`	C	Record type
		`(` *Type* `with` *Label FieldType* … *Label FieldType* `)`	D	Record extension
		`(` *Type* `where` *Label FieldType* … *Label FieldType* `)`	D	Record field override
		`\` $KindedPolarityId_1 \ldots KindedPolarityId_n$ `=` *Type*	C	Type operator ($n \geq 1$)
		`(` *Type* $Type_1 \ldots Type_n$ `)`	C	Type application ($n \geq 1$)
		`(` `rec` *KindedId* `=` *Type* `)`	C	Recursive type
FieldType	=	*Type*	C	Value field
		`#` *Id Constr*	C	Type field
RType	=	$\langle empty \rangle$	R	Omitted type annotation
		`:` *Type*	C	Explicit type annotation

3.3.9 Polarities

Polarity	=	⟨*empty*⟩	C	Mixed polarity
		`Pos`	C	Positive polarity
		`Neg`	C	Negative polarity
		`Con`	C	Constant polarity

3.3.10 Kinds

Kind	=	`(` *Polarity* $Kind_1$ ... *Polarity* $Kind_n$ `->` *Kind* `)`	C	Operator kind ($n \geq 1$)
		`Type`	C	Type kind
KindedPolarityId	=	*Id* `:` *Polarity Kind*	C	Explicitly-kinded identifier
		Id	D	Implicitly-kinded identifier

3.3.11 Labels

Label	=	⟨*empty*⟩	C	Anonymous label
		Id `=`	C	Explicit label

Chapter 4

Extra-Linguistic Features

There are a few constructs that appear in the Pict syntax but that are absent from the rest of this definition. We describe them informally here.

4.1 Separate Compilation

A Pict program consists of a collection of named *compilation units*, each comprising a sequence of `import` statements followed by a sequence of declarations. Individual units can be compiled separately, but for definitional purposes the whole collection is treated as a single declaration sequence, as follows:

- Begin with the compilation units that has been designated as the *main unit*.
- List the names of the compilation units accessible from the main unit by performing a preorder traversal of the DAG generated by the `import` statements at the head of each unit.
- In this list, eliminate all but the first occurrence of each name.
- Replace each name by the corresponding declaration sequence to form a single declaration sequence $\mathtt{d}_1 \ldots \mathtt{d}_n$.
- Add a null process at the end. The final process expression denoted by the program is $(\mathtt{d}_1 \ldots \mathtt{d}_n\ \mathtt{()})$.

If the graph generated by the `import` statements contains a cycle, or if the final declaration sequence contains two bindings for the same top-level identifier, the program is erroneous.

4.2 Inline C Code

A value of the form $(\mathtt{ccode\ a\ m\ "code"\ fv}_1 \ldots \mathtt{fv}_n)$ causes `code` to be included at this point in the intermediate C code passed to GCC. This facility is used in many of the Pict standard libraries to access C primitives and C library routines.

The precise behaviour of a piece of inline C code is highly dependant on the code generation strategy used in the Pict compiler. Code generation for Pict is the subject of current research, so it is almost certain that the interface between Pict and C will change in the future. In light of this, Pict users should attempt to minimise their usage of inline C code.

4.2.1 Accessing Arguments

The code string `code` can refer to its argument fields using embedded `#` symbols. For example, the `ccode` expression `(ccode 0 P "(# + #)" x y)` yields a piece of C code which adds (using the C addition operator) the Pict variables `x` and `y`. The i'th `#` symbol in the code string corresponds to the i'th argument given to the `ccode` expression. The number of arguments must match the number of `#` symbols in the code string (literal `#` symbols can be included in a code string using the escape sequence `##`).

4.2.2 Pure Expressions

The P flag in the above example indicates that the expression is a pure expression (and as such, may be moved or eliminated by the optimiser). Any ccode expressions with the flag P must denote a valid C *atomic* expression (which is why we explicitly parenthesized the addition operation in the above example).

4.2.3 Constant Expressions

Other flag values are possible. For example, the flag C in (ccode 0 C "NULL") tells the optimiser that the expression NULL is a C constant. This lets the optimiser know that it can use the ccode expression as part of the initialiser for a top-level constant (C forces us to make this distinction, since it disallows non-constant initialisers at the top-level). Any ccode expression with the C flag must denote a valid C *atomic* expression.

4.2.4 Converting Pict and C Values

Various C macros are available to help convert Pict values to C values and vice versa. Most importantly, the macro I converts a Pict integer into a C integer (this is currently implemented as a right-shift, since Pict uses the low-order bits of each value to indicate whether it is a pointer or not). The macro intInt converts a C integer into a Pict integer (note that we lose one bit of precision when doing this).

For example, (ccode 0 P "intInt(I(#) + I(#))" x y) converts the Pict integers x and y to C integers, adds them, and then converts the resulting value back into a Pict integer. (Note that this is not the optimal way to add two Pict integers, but it serves as a good example.)

The S macro converts Pict strings into C strings (without doing any allocation). Note, however that Pict strings are allowed to contain nulls (since we keep an explicit length field for each string). Passing such a string to a C function may not give the correct behaviour, since the C function may interpret the embedded null as the end of the string. Converting a C string into a Pict string is more complicated, since the C string must be copied into the Pict heap. The library function prim.fromCString does this.

4.2.5 Reading Mutable State

A ccode expression with the flag R may safely read from mutable storage. For example,

```
(ccode 0 R "intInt(errno)")
```

reads an integer from the global variable errno (and converts it to a Pict integer). The optimiser is more careful about where it moves C code with the R flag (in particular, it will not move such code past C code whichs modifies mutable storage). One important feature of the R flag is that it lets the optimiser know that it is safe to discard the ccode (if its result is never referenced).

4.2.6 Writing Mutable State

A ccode expression with the flag W may safely read and write mutable storage. For example,

```
(ccode 0 W "intInt(f(I(#)))" x)
```

converts the Pict integer x to a C integer, calls the C function f (which may read and write to various global variables, for example), and then converts the resulting integer back into a Pict integer. As with ccode that reads from mutable state, the optimiser is careful about where it moves ccode with the W flag. Note, however, that the optimiser cannot discards a piece of inline C code with the W flag, since it cannot be sure which parts of the program can observe the side-effects of the inline C code.

4.2.7 Code With No Result

The flag E is similar to the W flag since it allows a piece of ccode to read and write mutable storage. However, specifying the E flags indicates that the expression has no result value, and is to be evaluated for its side-effects only. Unlike all the previous ccode forms, ccode with the E flag must be a well-formed C *statement*, not a C expression. For example, `(ccode 0 E "fputs(S(#),stderr);" s)` converts the Pict string `s` to a C string and then calls the `fputs` function to print out the string. The optimiser never discards a piece of inline C code with the `E` flag. In Pict, the result type of such a piece of C code is always `[]`, the unit type.

4.2.8 Storage Allocation

The integer `a` in the expression (`ccode a m "code"` $\mathtt{fv}_1 \ldots \mathtt{fv}_n$) indicates how many words of storage are required by `code`. Pict processes do a single heap check at the start of each basic block, so there is no need to check for heap exhaustion in `code` itself. (A limitation of the current approach is that the storage required must be fixed at compile-time. We hope to fix this some time in the future.)

Chapter 5

Simple Derived Forms

Some very straightforward derived forms are desugared during parsing (i.e. before type reconstruction), avoiding the need to equip them with typing and type reconstruction rules. We write $\texttt{phrase}_1 \Rightarrow \texttt{phrase}_2$, pronounced "$\texttt{phrase}_1$ desugars to $\texttt{phrase}_2$."

5.1 Declarations

The short declaration form `v;` is replaced by a `val` binding with an empty record pattern.

$$\texttt{v; } \Rightarrow \texttt{ val [] = v} \tag{Tr-Semi}$$

A type abbreviation is replaced by a `val` declaration with an equality constraint in the pattern. Abbreviations for type operators are handled similarly.

$$\texttt{type X = T} \Rightarrow \texttt{val [\#X=T] = [\#T]} \tag{Tr-Abbr}$$

$$\begin{array}{c}\texttt{type (X } \texttt{Y}_1\texttt{:K}_1\texttt{...Y}_n\texttt{:K}_n\texttt{) = T} \\ \Rightarrow \texttt{ val [\#X = \textbackslash Y}_1\texttt{:K}_1\texttt{...Y}_n\texttt{:K}_n\texttt{=T] = [\#\textbackslash Y}_1\texttt{:K}_1\texttt{...Y}_n\texttt{:K}_n\texttt{=T]}\end{array} \tag{Tr-OpAbbr}$$

If a `def` declaration is preceded by the keyword `inline`, then uses of `x` later in the program will be compiled by expanding the definition in place instead of sending messages along channels. An `inline def` is not allowed to be recursively-defined, which is why we describe its meaning in terms of an anonymous abstraction. Inlining should be used with care, since it a large increase in the size of compiled code.

$$\texttt{inline def x a} \Rightarrow \texttt{val x = \textbackslash a} \tag{Tr-Inline}$$

5.2 Processes

The parallel composition of three or more processes is transformed to a nested sequence of binary parallel compositions. Declaration sequences are replaced by nested sequences of individual declarations.

$$\texttt{(e}_1 \texttt{ | ... | e}_n\texttt{)} \Rightarrow \texttt{(e}_1 \texttt{ | ... (e}_{n-1} \texttt{ | e}_n\texttt{))} \tag{Tr-MultiPrl}$$

$$\texttt{(d}_1\texttt{...d}_n \texttt{ e)} \Rightarrow \texttt{(d}_1 \texttt{ ... (d}_n \texttt{ e))} \tag{Tr-DecSeq}$$

5.3 Abstractions

A value abstraction is translated by adding the implicit result channel to the list of patterns already given and changing the body into a process that sends on this channel:

$$\texttt{(l}_1\texttt{fp}_1\texttt{...l}_n\texttt{fp}_n\texttt{) = v} \Rightarrow \texttt{[l}_1\texttt{fp}_1\texttt{...l}_n\texttt{fp}_n\texttt{ r] = r!v} \qquad \text{(Tr-VAbs-1)}$$

$$\texttt{(l}_1\texttt{fp}_1\texttt{...l}_n\texttt{fp}_n\texttt{):T = v} \Rightarrow \texttt{[l}_1\texttt{fp}_1\texttt{...l}_n\texttt{fp}_n\texttt{ r:/T] = r!v} \qquad \text{(Tr-VAbs-2)}$$

(The channel variable `r` is assumed to be different from the variables bound by the field patterns and the variables mentioned in `v`.)

5.4 Values

Declaration sequences in values are replaced by nested sequences of declarations.

$$\texttt{(d}_1\texttt{...d}_n\texttt{ v)} \Rightarrow \texttt{(d}_1\texttt{ ... (d}_n\texttt{ v))} \qquad \text{(Tr-DecSeq-V)}$$

The "folded application" operators `>` and `<` are expanded to nested sequences of applications.

$$\texttt{(v < v}_1\texttt{...v}_n\texttt{)} \Rightarrow \texttt{(v ... (v v}_1\texttt{ v}_2\texttt{) ... v}_n\texttt{)} \qquad \text{(Tr-ApplyL)}$$

$$\texttt{(v > v}_1\texttt{...v}_n\texttt{)} \Rightarrow \texttt{(v v}_1\texttt{ ... (v v}_{n-1}\texttt{ v}_n\texttt{) ...)} \qquad \text{(Tr-ApplyR)}$$

5.5 Kinded Identifiers

Where a kinded identifier is expected and the kind annotation is omitted, kind `Type` is assumed.

$$\texttt{X} \Rightarrow \texttt{X:Type} \quad \text{(in contexts expecting a kinded type id)} \qquad \text{(Tr-KindedId)}$$

5.6 Field Types

Type fields in record patterns and types may be given as bare variables; the associated constraint is taken to be `<Top` (i.e., variables without explicit constraints range over arbitrary types of kind `Type`).

$$\texttt{\#X} \Rightarrow \texttt{\#X<Top} \qquad \text{(Tr-EmptyConstr)}$$

Chapter 6

Variable Scoping

For simplicity, we separate the process of resolving variable scopes from that of typechecking. The scope resolution process yields an alpha-renamed copy of the original term. The alpha-renamed term has the property that every bound variable is unique (a property which is demanded by the typing rules we give later).

Given the variable scoping rules, it is easy to define the free and bound variables of a construct, written $FV(\cdot)$ and $BV(\cdot)$ respectively. Also, in a term with unique bound variables, it is easy to define the substitution of type $\mathtt{T}$ for type variable $\mathtt{X}$ (written $\{\mathtt{X} \mapsto \mathtt{T}\}\cdot$) and the substitution of value $\mathtt{v}$ for variable $\mathtt{x}$ (written $\{\mathtt{x} \mapsto \mathtt{v}\}\cdot$).

6.1 Judgement Forms

Strictly speaking, we should write $\Gamma \vdash \mathtt{e} \to \mathtt{e}'$ instead of $\Gamma \vdash \mathtt{e}$, and similarly for the other scope resolution judgements. However, since it is trivial to read off the alpha-renamed term from a scope resolution derivation, we leave it implicit to lighten the rules.

$\Gamma \vdash \mathtt{e}$	process expression $\mathtt{e}$ is well formed
$\Gamma \vdash \mathtt{d} \triangleright \Delta$	declaration $\mathtt{d}$ is well formed and yields bindings Δ
$\Gamma \vdash \mathtt{v}$	value is well formed
$\Gamma \vdash \mathtt{fv}$	field value $\mathtt{fv}$ is well formed
$\Gamma \vdash \mathtt{p} \triangleright \Delta$	pattern $\mathtt{p}$ is well formed and yields bindings Δ
$\Gamma \vdash \mathtt{fp} \triangleright \Delta; \Delta'$	field pattern $\mathtt{p}$ is well formed and yields type bindings Δ and pattern bindings Δ'
$\Gamma \vdash \mathtt{a}$	abstraction $\mathtt{a}$ is well formed
$\Gamma \vdash \mathtt{T}$	type $\mathtt{T}$ is well formed
$\Gamma \vdash \mathtt{RT}$	reconstructable type $\mathtt{RT}$ is well formed
$\Gamma \vdash \mathtt{FT} \triangleright \Delta$	field type $\mathtt{FT}$ is well formed and yields type bindings Δ

(Note that kinds have no scoping rules, since they contain no binding constructs and are always well-formed.)

6.2 Auxiliary Definitions

6.2.1 Field label checking

We require that all the (nonempty) labels in a record be distinct. The predicate $\mathit{distinct}(\mathtt{l}_1 ... \mathtt{l}_n)$ checks this.

6.3 Scoping Rules

6.3.1 Processes

The null process, parallel composition, output, and conditional process forms do not introduce any new variable bindings.

$$\Gamma \vdash \texttt{()} \tag{B-Null}$$

$$\frac{\Gamma \vdash \texttt{e}_1 \qquad \Gamma \vdash \texttt{e}_2}{\Gamma \vdash \texttt{(e}_1 \texttt{ | e}_2\texttt{)}} \tag{B-Prl}$$

$$\frac{\Gamma \vdash \texttt{v}_1 \qquad \Gamma \vdash \texttt{v}_2}{\Gamma \vdash \texttt{v}_1\texttt{!v}_2} \tag{B-Out}$$

$$\frac{\Gamma \vdash \texttt{b} \qquad \Gamma \vdash \texttt{e}_1 \qquad \Gamma \vdash \texttt{e}_2}{\Gamma \vdash \texttt{if b then e}_1 \texttt{ else e}_2} \tag{B-If}$$

An input process `v?a` does introduce some variable bindings, but they are handled in the scoping rules for abstractions.

$$\frac{\Gamma \vdash \texttt{v} \qquad \Gamma \vdash \texttt{a}}{\Gamma \vdash \texttt{v?a}} \tag{B-In}$$

A process preceded by a declaration is well formed if the declaration is well formed and the body is well formed in a context extended with the bindings introduced by the declaration.

$$\frac{\Gamma \vdash \texttt{d} \triangleright \Delta \qquad \Gamma, \Delta \vdash \texttt{e}}{\Gamma \vdash \texttt{(d e)}} \tag{B-Dec}$$

6.3.2 Declarations

A `new` declaration introduces a variable binding for the new channel identifier.

$$\frac{\Gamma \vdash \texttt{T}}{\Gamma \vdash \texttt{new x:T} \triangleright \texttt{x}} \tag{B-New}$$

A `run` declaration does not introduce any new variable bindings.

$$\frac{\Gamma \vdash \texttt{e}}{\Gamma \vdash \texttt{run e} \triangleright \bullet} \tag{B-Run}$$

A `val` declaration introduces whatever bindings are introduced by the pattern `p`.

$$\frac{\Gamma \vdash \texttt{v} \qquad \Gamma \vdash \texttt{p} \triangleright \Delta}{\Gamma \vdash \texttt{val p = v} \triangleright \Delta} \tag{B-Val}$$

A recursive definition introduces bindings $\texttt{x}_1, \ldots, \texttt{x}_n$. Each abstraction $\texttt{a}_i$ is scoped in the context Γ extended with the bindings $\texttt{x}_1, \ldots, \texttt{x}_n$, thereby allowing recursive references to the process definitions $\texttt{x}_1, \ldots, \texttt{x}_n$.

$$\frac{\Gamma, \texttt{x}_1, \ldots, \texttt{x}_n \vdash \texttt{a}_i \qquad \text{for each } i}{\Gamma \vdash \texttt{def x}_1\texttt{a}_1 \texttt{ ... and x}_n\texttt{a}_n \triangleright \texttt{x}_1, \ldots, \texttt{x}_n} \tag{B-Def}$$

6.3.3 Abstractions

In a simple process abstraction `p=e` the bindings introduced in `p` have scope `e`.

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \Delta \qquad \Gamma, \Delta \vdash \mathtt{e}}{\Gamma \vdash \mathtt{p=e}} \tag{B-Abs}$$

6.3.4 Patterns

A variable pattern introduces a binding for `x`.

$$\frac{\Gamma \vdash \mathtt{RT}}{\Gamma \vdash \mathtt{x:RT} \triangleright \mathtt{x}} \tag{BP-Var}$$

A wildcard pattern introduces no bindings.

$$\frac{\Gamma \vdash \mathtt{RT}}{\Gamma \vdash \mathtt{\_:RT} \triangleright \bullet} \tag{BP-Wild}$$

A layered pattern introduces a binding for `x`, plus whatever bindings are generated by the inner pattern `p`.

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \Delta \qquad \Gamma \vdash \mathtt{RT}}{\Gamma \vdash \mathtt{x:RT@p} \triangleright \mathtt{x}, \Delta} \tag{BP-As}$$

A rectype pattern does not bind any variables itself, so we simply return the variables bound by the inner pattern `p`.

$$\frac{\Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{p} \triangleright \Delta}{\Gamma \vdash \mathtt{(rec:RT\ p)} \triangleright \Delta} \tag{BP-Rec}$$

Each field in a record pattern may introduces type bindings which are in scope in any subsequent records fields (fields to the right of the current field). The pattern bindings Δ'_i (which are bindings that arise from sub-patterns), are not in scope in fields to the right of the current field.

$$\frac{\Gamma \vdash \mathtt{fp}_1 \triangleright \Delta_1; \Delta'_1 \quad \ldots \quad \Gamma, \Delta_1, \ldots, \Delta_{n-1} \vdash \mathtt{fp}_n \triangleright \Delta_n; \Delta'_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{[l_1 fp_1 \ldots l_n fp_n]} \triangleright \Delta_1, \Delta'_1, \ldots, \Delta_n, \Delta'_n} \tag{BP-Record}$$

6.3.5 Field Patterns

All of the judgement forms can be readily understood, except perhaps for the field pattern judgements. Rather than returning a single context as the other pattern judgement forms do, they return a pair of contexts. The first context contains the type bindings introduced by the field pattern (if any). These bindings are in scope in any field patterns to the right of the current field pattern. The second scoping context returned by the field pattern rules contains all the bindings introduced by sub-patterns, which are not in scope in fields to the right of the current field.

$$\frac{\Gamma \vdash \mathtt{p} \triangleright \Delta}{\Gamma \vdash \mathtt{p} \triangleright \bullet; \Delta} \tag{BFP-Val}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{\#X<T} \triangleright \mathtt{X}; \bullet} \tag{BFP-Bound}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{\#X=T} \triangleright \mathtt{X}; \bullet} \tag{BFP-Equal}$$

6.3.6 Paths

The following two rules formalise how we look up a variable x in the context (we simply discard entries from the end of the context until we find one which matches x).

$$\Gamma, \mathtt{x} \vdash \mathtt{x} \tag{BPath-Var-1}$$

$$\frac{\Gamma \vdash \mathtt{x} \qquad \mathtt{x} \neq \mathtt{y}}{\Gamma, \mathtt{y} \vdash \mathtt{x}} \tag{BPath-Var-2}$$

Field projection does not introduce any new bound variables.

$$\frac{\Gamma \vdash \mathtt{path}}{\Gamma \vdash \mathtt{path.x}_i} \tag{BPath-Label}$$

6.3.7 Values

A value preceded by a declaration is well formed if the declaration is well formed and the value is well formed in a context extended by the bindings generated by the declaration.

$$\frac{\Gamma \vdash \mathtt{d} \triangleright \Delta \qquad \Gamma, \Delta \vdash \mathtt{v}}{\Gamma \vdash \mathtt{(d\ v)}} \tag{BV-Dec}$$

An anonymous abstraction value does introduce some new bound variables, but these are dealt with in the scoping rules for abstractions.

$$\frac{\Gamma \vdash \mathtt{a}}{\Gamma \vdash \mathtt{\backslash a}} \tag{BV-AnonAbs}$$

The other forms of value do not introduce any new bound variables.

$$\Gamma \vdash \mathtt{k} \tag{BV-Const}$$

$$\frac{\Gamma \vdash \mathtt{b} \qquad \Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{v}_1 \qquad \Gamma \vdash \mathtt{v}_2}{\Gamma \vdash \mathtt{if{:}RT\ b\ then\ v_1\ else\ v_2}} \tag{BV-IfV}$$

$$\frac{\Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{v}}{\Gamma \vdash \mathtt{(rec{:}RT\ v)}} \tag{BV-Rec}$$

The record, `with` and application constructs introduce new labelled record fields, which we require to be pairwise distinct.

$$\frac{\Gamma \vdash \mathtt{fv}_1 \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{[l_1 fv_1 \ldots l_n fv_n]}} \tag{BV-Record}$$

$$\frac{\Gamma \vdash \mathtt{v} \qquad \Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{fv}_1 \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{(v{:}RT\ with\ l_1 fv_1 \ldots l_n fv_n)}} \tag{BV-With}$$

$$\frac{\Gamma \vdash \mathtt{v} \qquad \Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{fv}_1 \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{(v{:}RT\ l_1 fv_1 \ldots l_n fv_n)}} \tag{BV-App}$$

A `where` expression is only well-formed if the fields being replaced are explicitly-labeled value fields (the syntax for `where` is currently more permissive than this, since we hope to generalise the `where` operation in future versions of Pict).

$$\frac{\Gamma \vdash \mathtt{v} \qquad \Gamma \vdash \mathtt{RT} \qquad \Gamma \vdash \mathtt{v}_1 \quad \ldots \quad \Gamma \vdash \mathtt{v}_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{(v{:}RT\ where\ x_1{=}v_1 \ldots x_n{=}v_n)}} \tag{BV-Where}$$

6.3.8 Field Values

Field values do not introduce any new bound variables (value fields are ordinary values, for which no separate rule is needed).

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{\#T}} \qquad \text{(BFV-Type)}$$

6.3.9 Types

The following two rules formalise how we look up a type variable X in the context (we simply discard entries from the end of the context until we find one which matches X).

$$\Gamma, \mathtt{X} \vdash \mathtt{X} \qquad \text{(BT-TVar-1)}$$

$$\frac{\Gamma \vdash \mathtt{X} \qquad \mathtt{X} \neq \mathtt{Y}}{\Gamma, \mathtt{Y} \vdash \mathtt{X}} \qquad \text{(BT-TVar-2)}$$

Type operators introduce new bound variables whose scope is the body the the type operator.

$$\frac{\Gamma, \mathtt{X}_1, \ldots, \mathtt{X}_n \vdash \mathtt{T}}{\Gamma \vdash \mathtt{\backslash X}_1\mathtt{:K}_1 \ldots \mathtt{X}_n\mathtt{:K}_n\mathtt{=T}} \qquad \text{(BT-Abs)}$$

A recursive type `(rec X:K=T)` introduces a single bound variable X.

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{T}}{\Gamma \vdash \mathtt{(rec\ X:K=T)}} \qquad \text{(BT-Rec)}$$

The fields of a record type may introduce type bindings which are in scope in subsequent fields (to the right).

$$\frac{\Gamma \vdash \mathtt{FT}_1 \triangleright \Delta_1 \quad \ldots \quad \Gamma, \Delta_1, \ldots, \Delta_{n-1} \vdash \mathtt{FT}_n \triangleright \Delta_n}{\Gamma \vdash \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]}} \qquad \text{(BT-Record)}$$

All the remaining type constructors introduce no new bound variables.

$$\frac{\Gamma \vdash \mathtt{S} \qquad \Gamma \vdash \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash \mathtt{T}_n}{\Gamma \vdash \mathtt{(S\ T}_1 \ldots \mathtt{T}_n\mathtt{)}} \qquad \text{(BT-App)}$$

$$\Gamma \vdash \mathtt{Top} \qquad \text{(BT-Top)}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{\hat{}T}} \qquad \text{(BT-Chan)}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{?T}} \qquad \text{(BT-IChan)}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{!T}} \qquad \text{(BT-OChan)}$$

$$\frac{\Gamma \vdash \mathtt{T}}{\Gamma \vdash \mathtt{/T}} \qquad \text{(BT-RChan)}$$

$$\Gamma \vdash \texttt{Int} \qquad \text{(BT-Int)}$$

$$\Gamma \vdash \texttt{Char} \qquad \text{(BT-Char)}$$

$$\Gamma \vdash \texttt{Bool} \qquad \text{(BT-Bool)}$$

$$\Gamma \vdash \texttt{String} \qquad \text{(BT-String)}$$

6.3.10 Type Fields

An value type field does not introduce any type bindings.

$$\frac{\Gamma \vdash \texttt{T}}{\Gamma \vdash \texttt{T} \triangleright \bullet} \qquad \text{(BFT-Val)}$$

A type field introduces a binding for the type variable `X`.

$$\frac{\Gamma \vdash \texttt{T}}{\Gamma \vdash \texttt{\#X<T} \triangleright \texttt{X}} \qquad \text{(BFT-Bound)}$$

$$\frac{\Gamma \vdash \texttt{T}}{\Gamma \vdash \texttt{\#X=T} \triangleright \texttt{X}} \qquad \text{(BT-Equal)}$$

6.3.11 Reconstructable Types

A reconstructable type `RT` is either a type `T` (in which case the scoping rules for ordinary types apply), or it is empty (in which case it is trivially well-formed).

Chapter 7

Kinding

The kinding judgement and its associated auxiliary judgements are used to verify the well-formedness of type expressions.

7.1 Judgement Forms

The kinding and context well-formedness judgements are familiar from standard presentations of F^ω. The kinding judgement for record field types checks that value bindings are associated with types of kind `Type` and that type fields are associated with well-kinded constraints; in addition, it yields an environment Δ that records the constraints in type fields, for use in checking fields further to the right in the same record.

$\vdash \Gamma\ ok$	context Γ is well formed
$\Gamma \vdash \mathtt{T} \in \mathtt{K}$	type `T` has kind `K`
$\Gamma \vdash \mathtt{FT} \triangleright \Delta$	field type `FT` is well formed and yields constraints Δ

7.2 Auxiliary Definitions

We first define some notations that are used in the rules that follow.

7.2.1 Top at Higher Kinds

It is convenient to have a name for a maximal type in each kind. We write `Top:K` for the element of kind `K` defined as follows:

$$\begin{array}{lcl} \mathtt{Top{:}Type} & = & \mathtt{Top} \\ \mathtt{Top{:}(K_1\ldots K_n\text{->}K)} & = & \mathtt{\backslash X_1{:}K_1\ldots X_n{:}K_n\ =\ Top{:}K} \end{array}$$

7.2.2 Field Replacement

We write $\{\mathtt{x} \mapsto \mathtt{T}\}(\mathtt{x_1{=}T_1\ldots x_n{=}T_n})$ for the result of replacing the type of field `x` by `T` in the sequence of field types $\mathtt{x_1{=}T_1\ldots x_n{=}T_n}$. Note that field replacement is only defined when all fields in the sequence are value fields.

$$\begin{array}{rcll} \{\mathtt{x_1} \mapsto \mathtt{T}\}(\mathtt{x_1{=}T_1\ldots x_n{=}T_n}) & = & \mathtt{x_1{=}T\ \ x_2{=}T_2\ldots x_n{=}T_n} & \\ \{\mathtt{x} \mapsto \mathtt{T}\}(\mathtt{x_1{=}T_1\ldots x_n{=}T_n}) & = & \mathtt{x_1{=}T_1}\ \{\mathtt{x} \mapsto \mathtt{T}\}(\mathtt{x_2{=}T_2\ldots x_n{=}T_n}) & \text{if } \mathtt{x} \neq \mathtt{x_1} \end{array}$$

7.2.3 Conversion

Besides the usual rule of β-conversion on type expressions, we introduce three special rules: one for expanding abbreviations and two for simplifying record types. The rule R-Abbr replaces an abbreviation variable by its expansion. This requires that the context Γ be carried through the definition of the reduction relation, extending Γ with the abbreviation binders that are in scope at each point—so that, for example,

`[#X=Int ^X]` $=_\Gamma$ `[#X=Int ^Int]`. (See [PS96] for more details.) The rules for `with` and `where` types can be thought of as derived forms for these constructs. Like `with` and `where` on values, later rules (after this chapter) do not have to deal with them, but since they are type-directed they cannot be placed in Chapter 5.

$$(\backslash \mathtt{X}_1\mathtt{:K}_1\ldots\mathtt{X}_n\mathtt{:K}_n\mathtt{=T}\quad \mathtt{S}_1\ldots\mathtt{S}_n) \longrightarrow_\Gamma \{\mathtt{X}_1\mapsto\mathtt{S}_1\}\ldots\{\mathtt{X}_n\mapsto\mathtt{S}_n\}\mathtt{T} \tag{R-Beta}$$

$$\frac{\Gamma = \Gamma_1,\mathtt{X=T},\Gamma_2}{\mathtt{X}\longrightarrow_\Gamma \mathtt{T}} \tag{R-Abbr}$$

$$(\mathtt{[l_1FT_1\ldots l_mFT_m]}\ \mathtt{with}\ \mathtt{l}_{m+1}\mathtt{FT}_{m+1}\ldots\mathtt{l}_n\mathtt{FT}_n) \longrightarrow_\Gamma \mathtt{[l_1FT_1\ldots l_nFT_n]} \tag{R-With}$$

$$\begin{array}{c}(\mathtt{[l_1FT_1\ldots l_mFT_m]}\ \mathtt{where}\ \mathtt{x_1=T_1\ldots x_n=T_n})\\ \longrightarrow_\Gamma \mathtt{[}\{\mathtt{x}_1\mapsto\mathtt{T}_1\}\ldots\{\mathtt{x}_n\mapsto\mathtt{T}_n\}(\mathtt{l_1FT_1\ldots l_mFT_m})\mathtt{]}\end{array} \tag{R-Where}$$

Conversion, written $=_\Gamma$, is the transitive, reflexive, and symmetric closure of $\longrightarrow_\Gamma$.

7.3 Kinding Rules

7.3.1 Context well-formedness

A typing context is checked for well-formedness by checking that each of its entries is well kinded.

$$\vdash \bullet\ ok \tag{C-Empty}$$

$$\frac{\Gamma\vdash\mathtt{T}\in\mathtt{K}\qquad \mathtt{X}\notin dom(\Gamma)}{\vdash\Gamma,\mathtt{X<T}\ ok} \tag{C-TVar}$$

$$\frac{\Gamma\vdash\mathtt{T}\in\mathtt{K}\qquad \mathtt{X}\notin dom(\Gamma)}{\vdash\Gamma,\mathtt{X=T}\ ok} \tag{C-Abbr}$$

$$\frac{\Gamma\vdash\mathtt{T}\in\mathtt{Type}\qquad \mathtt{x}\notin dom(\Gamma)}{\vdash\Gamma,\mathtt{x:T}\ ok} \tag{C-Var}$$

The side conditions $\mathtt{X}\notin dom(\Gamma)$ and $\mathtt{x}\notin dom(\Gamma)$ ensure that all bound identifiers are distinct. This condition can always satisfied by transforming the program using the scope resolution rules (Chapter 6).

7.3.2 Kinding

The kind of a type variable `X` is the same as the kind of the constraint associated with `X` in the context.

$$\frac{\Gamma_1\vdash\mathtt{T}\in\mathtt{K}}{\Gamma_1,\mathtt{X<T},\Gamma_2\vdash\mathtt{X}\in\mathtt{K}} \tag{K-TVar}$$

$$\frac{\Gamma_1\vdash\mathtt{T}\in\mathtt{K}}{\Gamma_1,\mathtt{X=T},\Gamma_2\vdash\mathtt{X}\in\mathtt{K}} \tag{K-Abbr}$$

Top is the maximal element of kind `Type`.

$$\Gamma\vdash\mathtt{Top}\in\mathtt{Type} \tag{K-Top}$$

We check the well-formedness of $\backslash\mathtt{X}_1\mathtt{:K}_1\ldots\mathtt{X}_n\mathtt{:K}_n\mathtt{=T}$ by checking the kind of `T` in a context extended with the kinds of the bound variables $\mathtt{X}_1,\ldots,\mathtt{X}_n$ (the constraint $\mathtt{X}_i\mathtt{<Top:K}_i$ just records the kind of $\mathtt{X}_i$ since $\mathtt{Top:K}_i$ is the maximal element of kind $\mathtt{K}_i$).

$$\frac{\Gamma, \mathtt{X}_1\mathtt{<Top:K}_1, \ldots, \mathtt{X}_n\mathtt{<Top:K}_n \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\backslash X}_1\mathtt{:K}_1 \ldots \mathtt{X}_n\mathtt{:K}_n\mathtt{=T} \in \mathtt{(K}_1 \ldots \mathtt{K}_n\mathtt{->K)}} \tag{K-Abs}$$

The type application $\mathtt{(S\ T_1 \ldots T_n)}$ is well formed if `S` is a type operator of arity n and if the kinds of the arguments $\mathtt{T}_1, \ldots, \mathtt{T}_n$ match the kinds of the arguments to the type operator `S`.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{(K}_1 \ldots \mathtt{K}_n\mathtt{->K)} \qquad \Gamma \vdash \mathtt{T}_1 \in \mathtt{K}_1 \quad \ldots \quad \Gamma \vdash \mathtt{T}_n \in \mathtt{K}_n}{\Gamma \vdash \mathtt{(S\ T}_1 \ldots \mathtt{T}_n\mathtt{)} \in \mathtt{K}} \tag{K-App}$$

A recursive type `(rec X:K=T)` is well formed if `T` is well formed in a context extended with the type binding `X<Top:K`.

$$\frac{\Gamma, \mathtt{X<Top:K} \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{(rec\ X:K=T)} \in \mathtt{K}} \tag{K-Rec}$$

If types `^T`, `?T`, `!T`, and `/T` all have kind `Type` if `T` has kind `Type`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{\hat{}T} \in \mathtt{Type}} \tag{K-Chan}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{?T} \in \mathtt{Type}} \tag{K-IChan}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{!T} \in \mathtt{Type}} \tag{K-OChan}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{/T} \in \mathtt{Type}} \tag{K-RChan}$$

The types `Int`, `Char`, `Bool` and `String` all have kind `Type`.

$$\Gamma \vdash \mathtt{Int} \in \mathtt{Type} \tag{K-Int}$$

$$\Gamma \vdash \mathtt{Char} \in \mathtt{Type} \tag{K-Char}$$

$$\Gamma \vdash \mathtt{Bool} \in \mathtt{Type} \tag{K-Bool}$$

$$\Gamma \vdash \mathtt{String} \in \mathtt{Type} \tag{K-String}$$

7.3.3 Record Kinding

The only complicated kinding rules are those for record types, where—because type fields in records create bindings whose scope is the rest of the fields to the right—the kinding relation for fields must return an updated environment that for successive fields.

$$\frac{\Gamma \vdash \mathtt{FT}_1 \rhd \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash \mathtt{FT}_n \rhd \Delta_n}{\Gamma \vdash \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]} \in \mathtt{Type}} \tag{K-Record}$$

The kinding rule for `with` ensures that the type being extended is convertible to a record type. Moreover, the kinding of the additional bindings in a `with` type must not depend on bindings introduced by the existing fields.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \mathtt{S} =_\Gamma \mathtt{[l_1FT_1 \ldots l_mFT_m]} \qquad \Gamma \vdash \mathtt{FT}_{m+1} \rhd \Delta_{m+1} \quad \ldots \quad \Delta_{n-1} \vdash \mathtt{FT}_n \rhd \Delta_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)}{\Gamma \vdash \mathtt{(S\ with\ l}_{m+1}\mathtt{FT}_{m+1} \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{)} \in \mathtt{Type}} \tag{K-With}$$

The kinding rule for `where` ensures that the type being overwritten is convertible to a record type. The overriding bindings must all have kind `Type`, and the labels of the fields being overwritten must be a subset of the labels in the original record type.

$$\frac{\{\mathtt{x}'_1 \ldots \mathtt{x}'_n\} \subseteq \{\mathtt{x}_1 \ldots \mathtt{x}_m\} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \mathtt{S} =_\Gamma \mathtt{[x_1=T_1 \ldots x_m=T_m]} \qquad \Gamma \vdash \mathtt{T}'_1 \in \mathtt{Type} \quad \ldots \quad \Gamma \vdash \mathtt{T}'_n \in \mathtt{Type}}{\Gamma \vdash \mathtt{(S\ where\ x}'_1\mathtt{=T}'_1 \ldots \mathtt{x}'_n\mathtt{=T}'_n\mathtt{)} \in \mathtt{Type}} \tag{K-Where}$$

(Note that the scope resolution rules in Chapter 6 impose some additional constraints on labels, since we wish all labels in records to be distinct.)

7.3.4 Field Kinding

The kinding rules for fields check that their bindings are well-kinded and, for type bindings, return an environment recording the binding for later use.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{T} \rhd \Gamma} \tag{FK-Val}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\#X<T} \rhd \Gamma, \mathtt{X<T}} \tag{FK-Bound}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\#X=T} \rhd \Gamma, \mathtt{X=T}} \tag{FK-Equal}$$

Chapter 8

Subtyping

The subtyping relation determines when a value may be used in a context expecting a value with a "less informative" type.

8.1 Judgement Forms

The main subtyping judgement, $\Gamma \vdash \mathtt{S} < \mathtt{T}$, is familiar from standard presentations of $F^{\omega}_{\leq}$ (e.g., [Car90, PS96]). The auxiliary field subtyping judgement is based on the same relation, but also records the bindings associated with type fields (for later use in checking the subtyping of fields further to the right).

$\Gamma \vdash \mathtt{S} < \mathtt{T}$	S is a subtype of T
$\Gamma \vdash \mathtt{FT} < \mathtt{FT}' \rhd \Delta$	field type FT is a subtype of $\mathtt{FT}'$ and yields constraints Δ

8.2 Subtyping Rules

8.2.1 General Rules

The subtyping relation is transitive and includes the conversion relation (making it, in particular, reflexive and closed under expansion of abbreviations).

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{U} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} < \mathtt{U} \qquad \Gamma \vdash \mathtt{U} < \mathtt{T}}{\Gamma \vdash \mathtt{S} < \mathtt{T}} \tag{S-Trans}$$

$$\frac{\mathtt{S} =_{\Gamma} \mathtt{T}}{\Gamma \vdash \mathtt{S} < \mathtt{T}} \tag{S-Conv}$$

8.2.2 Type Variables

Top is a supertype of every type.

$$\Gamma \vdash \mathtt{S} < \mathtt{Top} \tag{S-Top}$$

8.2.3 Type Variables

Type variables are subtypes of their declared bounds. (Variables associated with equality constraints rather than subtyping constraints are handled by the S-Conv rule.)

$$\Gamma_1, \mathtt{X{<}T}, \Gamma_2 \vdash \mathtt{X} < \mathtt{T} \tag{S-TVar}$$

8.2.4 Channel Types

Pict's four basic channel types are classified as follows: `^T` (the type of input/output channels carrying elements of `T`) is a subtype of both `!T` (output channels accepting `T`) and `?T` (input channels yielding `T`). Also `/T` (responsive output channels carrying `T`) is a subtype of `!T`.

$$\Gamma \vdash \texttt{\^{}T} < \texttt{?T} \qquad \text{(S-ChanIChan)}$$

$$\Gamma \vdash \texttt{\^{}T} < \texttt{!T} \qquad \text{(S-ChanOChan)}$$

$$\Gamma \vdash \texttt{/T} < \texttt{!T} \qquad \text{(S-RChanOChan)}$$

The operator `?` is covariant; `!` and `/` are contravariant; `^` is invariant (and hence needs no subtyping rule of its own).

$$\frac{\Gamma \vdash \texttt{S} < \texttt{T}}{\Gamma \vdash \texttt{?S} < \texttt{?T}} \qquad \text{(S-IChan)}$$

$$\frac{\Gamma \vdash \texttt{T} < \texttt{S}}{\Gamma \vdash \texttt{!S} < \texttt{!T}} \qquad \text{(S-OChan)}$$

$$\frac{\Gamma \vdash \texttt{T} < \texttt{S}}{\Gamma \vdash \texttt{/S} < \texttt{/T}} \qquad \text{(S-RChan)}$$

8.2.5 Basic Types

Characters can be implicitly coerced to integers.

$$\Gamma \vdash \texttt{Char} < \texttt{Int} \qquad \text{(S-CharInt)}$$

8.2.6 Abstraction and Application

The subtype relation is extended pointwise to higher types.

$$\frac{\Gamma, \texttt{X}_1\texttt{<Top:K}_1, \ldots, \texttt{X}_n\texttt{<Top:K}_n \vdash \texttt{S} < \texttt{T}}{\Gamma \vdash \texttt{\textbackslash X}_1\texttt{:K}_1 \ldots \texttt{X}_n\texttt{:K}_n\texttt{=S} < \texttt{\textbackslash X}_1\texttt{:K}_1 \ldots \texttt{X}_n\texttt{:K}_n\texttt{=T}} \qquad \text{(S-Abs)}$$

$$\frac{\Gamma \vdash \texttt{S} < \texttt{T}}{\Gamma \vdash \texttt{(S U}_1 \ldots \texttt{U}_n\texttt{)} < \texttt{(T U}_1 \ldots \texttt{U}_n\texttt{)}} \qquad \text{(S-App)}$$

Note that S-App requires that the arguments to the two type operators be syntactically identical. This requirement can be satisfied by using S-Conv and S-Trans to place the arguments in this form, if necessary, before applying S-App. (The same consideration applies at many other points in this chapter, e.g. in S-Equal, S-Equal-Bound, S-ChanIChan, etc.)

8.2.7 Recursive Types

For subtyping recursive types, we use the "Amber rule" (cf. [AC93]): we assume that the bound variables representing the two types are in the subtype relation and then try to prove that the whole types are in the subtype relation.

$$\frac{\Gamma, \texttt{Y<Top:K}, \texttt{X<Y} \vdash \texttt{S} < \texttt{T}}{\Gamma \vdash \texttt{(rec X:K=S)} < \texttt{(rec Y:K=T)}} \qquad \text{(S-Rec)}$$

8.2.8 Record Types

In subtyping record types, we allow new fields to be added on the right and we allow the types of common fields to be refined. We use a separate field subtyping relation, since there are several kinds of fields to take into account. Also, type fields generate constraints that may be needed for reasoning about fields further to the right; an updated environment containing these constraints is returned by the field subtyping judgement.

$$\frac{\Gamma \vdash \mathtt{FT}_1 < \mathtt{FT}'_1 \rhd \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash \mathtt{FT}_n < \mathtt{FT}'_n \rhd \Delta_n}{\Gamma \vdash \mathtt{[l_1FT_1 \ldots l_nFT_n \ldots]} < \mathtt{[l_1FT'_1 \ldots l_nFT'_n]}} \tag{S-Record}$$

8.2.9 Record Fields

For value fields, we simply invoke the ordinary subtype relation covariantly.

$$\frac{\Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{S} < \mathtt{T} \rhd \Gamma} \tag{S-Val}$$

For type fields with bounds, we use an analog of the subtyping rule for bounded existentials.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{\#X{<}S} < \mathtt{\#X{<}T} \rhd \Gamma, \mathtt{X{<}S}} \tag{S-Bound}$$

In fields with equality constraints, the constraints must be equal.

$$\Gamma \vdash \mathtt{\#X{=}T} < \mathtt{\#X{=}T} \rhd \Gamma, \mathtt{X{=}T} \tag{S-Equal}$$

Finally, an equality constraint may be promoted to a subtyping constraint.

$$\Gamma \vdash \mathtt{\#X{=}T} < \mathtt{\#X{<}T} \rhd \Gamma, \mathtt{X{=}T} \tag{S-Equal-Bound}$$

Chapter 9

Typing

This chapter formalizes the well-formedness judgement for process expressions and several associated auxiliary judgement forms. We give here a declarative formulation that applies only to programs where the programmer has provided explicit type information. A more algorithmic presentation that also handles omitted type annotations is given in Chapter 10. (It is the latter that is actually implemented in the Pict compiler: the more declarative presentation in this chapter is offered simply for documentation.)

9.1 Judgement Forms

The typing judgements for different syntactic categories all have slightly different forms, reflecting the different kinds of information that must be "passed back" to the enclosing context. For a process expression, we simply verify that the constraints imposed by the types of free variables are respected. For a declaration, we check internal consistency and also return an environment giving the types of the variables that it introduces. For a value, we calculate its own type. For a field value (which may be either a value or a type) we calculate a corresponding field type. For a pattern or record field pattern, we check the type of values that it can match and also return an environment that records the bindings it creates. For an abstraction, we check what type of value it expects.

$\Gamma \vdash \mathtt{e}\ ok$	process expression `e` is well-formed
$\Gamma \vdash \mathtt{d} \triangleright \Delta$	declaration `d` is well formed and yields bindings Δ
$\Gamma \vdash \mathtt{v} \in \mathtt{T}$	value (or path) `v` has type `T` under assumptions Γ
$\Gamma \vdash \mathtt{fv} \in \mathtt{FT}$	field value `fv` has field type `FT`
$\Gamma \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta$	pattern `p` requires type `T` and yields bindings Δ
$\Gamma \vdash \mathtt{fp} \in \mathtt{FT} \triangleright \Delta$	field pattern `p` requires field type `FT` and yields bindings Δ
$\Gamma \vdash \mathtt{a} \in \mathtt{T}$	abstraction `a` is well formed and accepts type `T`

9.2 Auxiliary Definitions

9.2.1 Typing of Constants

The helper function *TypeOfConst* defines the obvious mapping from constant values—integers, strings, characters, and the boolean constants `true` and `false`—to their types (`Int`, `String`, `Char`, or `Bool`).

9.2.2 Unrolling of Recursive Types

The typing rules for `rec` values and patterns involve "unrolling" a recursive type to reveal its body. For recursive types of kind `Type`, this is the standard operation of replacing instances of the recursively bound variable by copies of the whole recursive type. For recursive types of higher kinds, it may be necessary to unroll "under type applications."

$$\frac{\mathtt{S} =_\Gamma \mathtt{(rec\ X{:}K\ =\ T)}}{\Gamma \vdash \mathtt{S} \leadsto \{\mathtt{X} \mapsto \mathtt{(rec\ X{:}K\ =\ T)}\}\mathtt{T}} \qquad (\textsc{Unroll-Rec})$$

$$\frac{\mathtt{S} =_\Gamma \mathtt{(T\ T_1 \ldots T_n)} \qquad \Gamma \vdash \mathtt{T} \leadsto \mathtt{U}}{\Gamma \vdash \mathtt{S} \leadsto \mathtt{(U\ T_1 \ldots T_n)}} \qquad (\textsc{Unroll-App})$$

9.3 Typing Rules

Note that the rule of *subsumption* usually found in type systems with subtype relations is missing here; instead, some of the rules below include premises of the form $\Gamma \vdash \mathtt{v} \in \mathtt{S} < \mathtt{T}$, abbreviating pairs of premises $\Gamma \vdash \mathtt{v} \in \mathtt{S}$ and $\Gamma \vdash \mathtt{S} < \mathtt{T}$.

9.3.1 Processes

The null process is well typed in any well-formed context.

$$\frac{\vdash \Gamma\ \mathit{ok}}{\Gamma \vdash \mathtt{()}\ \mathit{ok}} \qquad (\textsc{E-Null})$$

The parallel composition of two processes is well formed if each is well formed separately.

$$\frac{\Gamma \vdash \mathtt{e}_1\ \mathit{ok} \qquad \Gamma \vdash \mathtt{e}_2\ \mathit{ok}}{\Gamma \vdash \mathtt{(e_1\ \mid\ e_2)}\ \mathit{ok}} \qquad (\textsc{E-Prl})$$

An output process $\mathtt{v_1!v_2}$ is well formed if $\mathtt{v}_1$'s type can be promoted to an output channel type $\mathtt{!T_2}$ and if the type of the argument $\mathtt{v}_2$ is a subtype of $\mathtt{T}_2$.

$$\frac{\Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 < \mathtt{!T_2} \qquad \Gamma \vdash \mathtt{v}_2 \in \mathtt{T}_2}{\Gamma \vdash \mathtt{v_1!v_2}\ \mathit{ok}} \qquad (\textsc{E-Out})$$

Similarly, an input process `v?a` is well formed if `v`'s type can be promoted to some input channel type `?S` and if the abstraction `a` can accept a value of type `S`.

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{T} < \mathtt{?S} \qquad \Gamma \vdash \mathtt{a} \in \mathtt{S}}{\Gamma \vdash \mathtt{v?a}\ \mathit{ok}} \qquad (\textsc{E-In})$$

A conditional process is well formed if the type of the guard is a subtype of `Bool` and the branches are both well formed.

$$\frac{\Gamma \vdash \mathtt{b} \in \mathtt{T} < \mathtt{Bool} \qquad \Gamma \vdash \mathtt{e}_1\ \mathit{ok} \qquad \Gamma \vdash \mathtt{e}_2\ \mathit{ok}}{\Gamma \vdash \mathtt{if\ b\ then\ e_1\ else\ e_2}\ \mathit{ok}} \qquad (\textsc{E-If})$$

A process preceded by a declaration is well formed if the declaration is well formed and the body is well formed under the bindings generated by the declaration.

$$\frac{\Gamma \vdash \mathtt{d} \rhd \Delta \qquad \Delta \vdash \mathtt{e}\ \mathit{ok}}{\Gamma \vdash \mathtt{(d\ e)}\ \mathit{ok}} \qquad (\textsc{E-Dec})$$

9.3.2 Declarations

A `new` declaration is well formed if the declared type for the new channel is indeed a channel type. A binding is generated for the new channel identifier.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} =_{\Gamma} \mathtt{\hat{}U}}{\Gamma \vdash \mathtt{new\ x:T} \rhd \Gamma, \mathtt{x:T}} \qquad \text{(D-NEW)}$$

A `run` declaration is well formed (and generates no bindings) if its body is.

$$\frac{\Gamma \vdash \mathtt{e}\ ok}{\Gamma \vdash \mathtt{run\ e} \rhd \Gamma} \qquad \text{(D-RUN)}$$

A value declaration is well formed if the type of the value is a subtype of the type accepted by the pattern. The bindings generated by the declaration are those generated by the pattern.

$$\frac{\Gamma \vdash \mathtt{p} \in \mathtt{T} \rhd \Delta \qquad \Gamma \vdash \mathtt{v} \in \mathtt{S} < \mathtt{T}}{\Gamma \vdash \mathtt{val\ p\ =\ v} \rhd \Delta} \qquad \text{(D-VAL)}$$

A recursive definition is well formed if each abstraction $\mathtt{a}_i$ can accept an argument of type $\mathtt{T}_i$. The context in which the abstractions are typed includes type bindings for all the recursively-defined identifiers $\mathtt{x}_i$. Each $\mathtt{x}_i$ is bound to the responsive channel types $\mathtt{/T}_i$. (In the typechecking algorithm, the $\mathtt{T}_i$ are calculated from the patterns in the abstractions, rather than being pulled from thin air, as this presentation of the rule does.)

$$\frac{\Gamma, \mathtt{x}_1\mathtt{:/T}_1, \ldots, \mathtt{x}_n\mathtt{:/T}_n \vdash \mathtt{a}_i \in \mathtt{T}_i \qquad \text{for each } i}{\Gamma \vdash \mathtt{def\ x}_1\mathtt{a}_1\ \mathtt{...\ and\ x}_n\mathtt{a}_n \rhd \mathtt{x}_1\mathtt{:/T}_1, \ldots, \mathtt{x}_n\mathtt{:/T}_n} \qquad \text{(D-DEF)}$$

9.3.3 Abstractions

A simple process abstraction `p=e` accepts whatever type is accepted by the pattern `p`, so long as the body `e` is well formed under the bindings generated by `p`.

$$\frac{\Gamma \vdash \mathtt{p} \in \mathtt{T} \rhd \Delta \qquad \Delta \vdash \mathtt{e}\ ok}{\Gamma \vdash \mathtt{p=e} \in \mathtt{T}} \qquad \text{(A-ABS)}$$

9.3.4 Patterns

A variable pattern accepts whatever type is explicitly declared for the variable and generates an appropriate binding.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{x:T} \in \mathtt{T} \rhd \Gamma, \mathtt{x:T}} \qquad \text{(P-VAR)}$$

A wildcard pattern matches whatever type is declared and generates no bindings.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{\_:T} \in \mathtt{T} \rhd \Gamma} \qquad \text{(P-WILD)}$$

A layered pattern matches whatever type `T` is declared and generates a binding for the whole matched value plus whatever bindings are generated by the inner pattern `p` (the type accepted by `p` may be a supertype of `T`).

$$\frac{\Gamma \vdash \mathtt{p} \in \mathtt{S} \rhd \Delta \qquad \Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash \mathtt{x:T@p} \in \mathtt{T} \rhd \Delta, \mathtt{x:T}} \qquad \text{(P-AS)}$$

A rectype pattern `(rec:S p)` accepts a value of the recursive type `S` if the inner pattern `p` accepts a value of type `T`, where `T` is a supertype of the result of "unrolling" `S` once.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \mathtt{S} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash \mathtt{p} \in \mathtt{T} \triangleright \Delta \qquad \Gamma \vdash \mathtt{U} < \mathtt{T}}{\Gamma \vdash \mathtt{(rec:S\ p)} \in \mathtt{S} \triangleright \Delta} \tag{P-Rec}$$

A record pattern matches a value of record type, where the types of the fields correspond to the types matched by the fields of the pattern. Any type fields will generate constraints that affect the typing of the remaining fields.

$$\frac{\Gamma \vdash \mathtt{fp}_1 \in \mathtt{FT}_1 \triangleright \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash \mathtt{fp}_n \in \mathtt{FT}_n \triangleright \Delta_n}{\Gamma \vdash \mathtt{[l_1fp_1 \ldots l_nfp_n]} \in \mathtt{[l_1FT_1 \ldots l_nFT_n]} \triangleright \Delta_n} \tag{P-Record}$$

9.3.5 Field Patterns

A field pattern can match either a value or a type field. Patterns for value fields are ordinary patterns, for which no separate rule is needed. Patterns for type fields introduce a type variable and either a subtyping or an equality constraint.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\#X<T} \in \mathtt{\#X<T} \triangleright \Gamma, \mathtt{X<T}} \tag{FP-Bound}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\#X=T} \in \mathtt{\#X=T} \triangleright \Gamma, \mathtt{X=T}} \tag{FP-Equal}$$

9.3.6 Paths

A path is a value consisting of a variable followed by a sequence of field projections. In the base case, a simple variable has whatever type is declared for it in the context.

$$\Gamma_1, \mathtt{x:T}, \Gamma_2 \vdash \mathtt{x} \in \mathtt{T} \tag{Path-Var}$$

A field projection of the form $\mathtt{path.x}_i$ has the type of the field labeled $\mathtt{x}_i$ in the type of `path`, which must be a record type. To simplify the formalization of the type system, we require that the type of `path` should not contain any type fields.

$$\frac{\Gamma \vdash \mathtt{path} \in \mathtt{S} < \mathtt{[l_1T_1 \ldots x_i{=}T_i \ldots l_nT_n]}}{\Gamma \vdash \mathtt{path.x}_i \in \mathtt{T}_i} \tag{Path-Label}$$

9.3.7 Values

A constant value `k` has type `Int`, `String`, `Char`, or `Bool`, as appropriate.

$$\Gamma \vdash \mathtt{k} \in \mathit{TypeOfConst}(\mathtt{k}) \tag{V-Const}$$

A value preceded by a declaration has the type of the body, which is typed in a context extended by the bindings generated by the declaration.

$$\frac{\Gamma \vdash \mathtt{d} \triangleright \Delta \qquad \Delta \vdash \mathtt{v} \in \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{(d\ v)} \in \mathtt{T}} \tag{V-Dec}$$

A conditional value has the type of the branches (which must have the same type), provided that the guard's type is a subtype of `Bool`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{b} \in \mathtt{S} < \mathtt{Bool} \qquad \Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 < \mathtt{T} \qquad \Gamma \vdash \mathtt{v}_2 \in \mathtt{T}_2 < \mathtt{T}}{\Gamma \vdash \mathtt{if:T\ b\ then\ v}_1\ \mathtt{else\ v}_2 \in \mathtt{T}} \qquad \text{(V-IfV)}$$

If `a` is an abstraction accepting a value of type `T`, then the "anonymous abstraction value" `\a` has type `/T` (a responsive channel carrying elements of `T`).

$$\frac{\Gamma \vdash \mathtt{a} \in \mathtt{T}}{\Gamma \vdash \mathtt{\backslash a} \in \mathtt{/T}} \qquad \text{(V-AnonAbs)}$$

A recursive value `(rec:T v)` has the declared recursive type `T` if the type of its body `v` is a subtype of the type obtained by unrolling `T` once.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash \mathtt{v} \in \mathtt{S} < \mathtt{U}}{\Gamma \vdash \mathtt{(rec:T\ v)} \in \mathtt{T}} \qquad \text{(V-Rec)}$$

A record value inhabits the record type corresponding to the types of its fields.

$$\frac{\Gamma \vdash \mathtt{fv}_1 \in \mathtt{FT}_1 \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \in \mathtt{FT}_n}{\Gamma \vdash \mathtt{[l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n\mathtt{]} \in \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]}} \qquad \text{(V-Record)}$$

The typing of `with` values is similar, except that the new fields are added to the fields of an existing record type.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \mathtt{S} =_\Gamma \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]} \qquad \Gamma \vdash \mathtt{v} \in \mathtt{T} < \mathtt{S} \\ \Gamma \vdash \mathtt{fv}_{m+1} \in \mathtt{FT}_{m+1} \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \in \mathtt{FT}_n \qquad \mathit{distinct}(\mathtt{l}_1 \ldots \mathtt{l}_n)\end{array}}{\Gamma \vdash \mathtt{(v:S\ with\ l}_{m+1}\mathtt{fv}_{m+1} \ldots \mathtt{l}_n\mathtt{fv}_n\mathtt{)} \in \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]}} \qquad \text{(V-With)}$$

A `where` value replaces the fields $\mathtt{x}_1, \ldots, \mathtt{x}_n$ in `v` with value fields of type $\mathtt{T}_1, \ldots, \mathtt{T}_m$.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \mathtt{S} =_\Gamma \mathtt{[l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]} \qquad \Gamma \vdash \mathtt{v} \in \mathtt{T} < \mathtt{S} \\ \Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash \mathtt{v}_m \in \mathtt{T}_m\end{array}}{\Gamma \vdash \mathtt{(v:S\ where\ x}_1\mathtt{=v}_1 \ldots \mathtt{x}_m\mathtt{=v}_m\mathtt{)} \in \mathtt{[}\{\mathtt{x}_1 \mapsto \mathtt{T}_1\} \ldots \{\mathtt{x}_m \mapsto \mathtt{T}_m\}\mathtt{l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\mathtt{]}} \qquad \text{(V-Where)}$$

The application rule is essentially a combination of a record construction and output rules. The result type of the application is given by the explicit type annotation `T`.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{v} \in \mathtt{S} < \mathtt{![l}_1\mathtt{FT}_1 \ldots \mathtt{l}_n\mathtt{FT}_n\ \mathtt{/T]} \\ \Gamma \vdash \mathtt{fv}_1 \in \mathtt{FT}_1 \quad \ldots \quad \Gamma \vdash \mathtt{fv}_n \in \mathtt{FT}_n\end{array}}{\Gamma \vdash \mathtt{(v:T\ l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n\mathtt{)} \in \mathtt{T}} \qquad \text{(V-App)}$$

9.3.8 Field Values

A value field has the type of the value and generates no type constraints.

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{T}}{\Gamma \vdash \mathtt{v} \in \mathtt{T}} \qquad \text{(FV-Val)}$$

A type field `#T` has type `#X=T`. (Note that the name of the bound variable `X` is "guessed" by this rule. In fact, the name will either be immaterial—when the field appears in an ordinary record construction—or else it will be determined by the context, by the type of the "function part" of an application expression.)

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash \mathtt{\#T} \in \mathtt{\#X=T}} \qquad \text{(FV-Type)}$$

Chapter 10

Type Reconstruction

This chapter formalizes the type reconstruction algorithm used in the Pict typechecker, which has the job of filling in type annotations omitted by the programmer. The algorithm relies on two basic techniques for filling in omitted information:

1. Instead of always working "from the bottom up"—synthesizing the types of phrases from the types of their subphrases—we use the fact that, in some contexts, we may know what type a phrase is *expected* to possess. In this case, we can fill in omitted type annotations by examining the expected type.

   To formalize this mechanism, we split most of the typing judgements into two forms:

   - a synthesis form (written, for example, $\Gamma \vdash \mathtt{v} \overrightarrow{\in} \mathtt{T}$) where the goal is to calculate a type T for the phrase v (the same one that would be assigned by the typing rules in Chapter 9); and
   - a checking form (written $\Gamma \vdash \mathtt{v} \overleftarrow{\in} \mathtt{T}$), where the goal is to show that the concrete type of v is some subtype of the expected type T.

   Intuitively, the arrow above the $\in$ symbol determines the direction of information flow: in a synthesis context, the type is determined by the annotations in the value; in a checking context, annotations may be determined by the expected type.

2. In a record value (or, similarly but more importantly, an application), omitted type fields can sometimes be recovered uniquely by examining the types of fields further to the right.

10.1 Auxiliary Definitions

10.1.1 Promotion

In many of the rules below, we shall need a name for "the smallest supertype of a given type T which does not have a type variiable or type application at its head." For example, in the typing rule E-Out for output processes $\mathtt{v}_1\mathtt{!}\mathtt{v}_2$, we need to find the type $\mathtt{T}_1$ of $\mathtt{v}_1$ and then calculate the smallest supertype of $\mathtt{T}_1$ that starts explicitly with an output channel constructor.

To promote a type variable X, we simply look up the constraint associated with X in the context, and promote that.

$$\frac{\Gamma = \Gamma_1, \mathtt{X<S}, \Gamma_2 \qquad \mathtt{S} \Uparrow_\Gamma \mathtt{T}}{\mathtt{X} \Uparrow_\Gamma \mathtt{T}} \qquad \text{(Promote-Var-1)}$$

$$\frac{\Gamma = \Gamma_1, \mathtt{X=S}, \Gamma_2 \qquad \mathtt{S} \Uparrow_\Gamma \mathtt{T}}{\mathtt{X} \Uparrow_\Gamma \mathtt{T}} \qquad \text{(Promote-Var-2)}$$

To promote a type application expression, we first try and promote the operator expression to an explicit type operator. If we succeed, we substitute the argument types $\mathtt{S}_1, \ldots, \mathtt{S}_n$ for the operator's bound variables, and then promote the body:

$$\frac{\mathtt{S} \Uparrow_\Gamma \mathtt{\backslash X_1{:}K_1 \ldots X_n{:}K_n{=}T} \qquad \{\mathtt{X}_1 \mapsto \mathtt{S}_1\}\ldots\{\mathtt{X}_n \mapsto \mathtt{S}_n\}\mathtt{T} \Uparrow_\Gamma \mathtt{U}}{(\mathtt{S}\ \mathtt{S}_1 \ldots \mathtt{S}_n) \Uparrow_\Gamma \mathtt{U}} \qquad \text{(Promote-App-1)}$$

If promoting the operator expression `S` does not yield an explicit type operator, then we have no chance of arriving at a type whose outermost type consructor is not an application expression, and we simply return the type unmodified.

$$\frac{\mathtt{S} \Uparrow_\Gamma \mathtt{T} \qquad \mathtt{T} \text{ is not a type operator}}{(\mathtt{S}\ \mathtt{S}_1 \ldots \mathtt{S}_n) \Uparrow_\Gamma (\mathtt{S}\ \mathtt{S}_1 \ldots \mathtt{S}_n)} \qquad \text{(Promote-App-2)}$$

Promotion has no effect on other type constructors:

$$\frac{\mathtt{S} \text{ is not a type variable or application expression}}{\mathtt{S} \Uparrow_\Gamma \mathtt{S}} \qquad \text{(Promote-Done)}$$

Note that the relation $\mathtt{S} \Uparrow_\Gamma \mathtt{T}$ is a total function on well-kinded types. That is, if $\mathtt{S} \Uparrow_\Gamma \mathtt{T}$ and $\mathtt{S} \Uparrow_\Gamma \mathtt{U}$, then $\mathtt{T} = \mathtt{U}$.

In a few places, we need to promote a type until it has a particular channel type constructor at its head. Since the channel type constructors themselves have subtyping behaviour, it is possible that we may need to do some extra promotion (after making use of the standard promotion relation).

$$\frac{\mathtt{S} \Uparrow_\Gamma \mathtt{\hat{}T} \quad \text{or} \quad \mathtt{S} \Uparrow_\Gamma \mathtt{?T}}{\mathtt{S} \Uparrow_\Gamma^{?} \mathtt{?T}} \qquad \text{(Promote-IChan)}$$

$$\frac{\mathtt{S} \Uparrow_\Gamma \mathtt{\hat{}T} \quad \text{or} \quad \mathtt{S} \Uparrow_\Gamma \mathtt{!T}}{\mathtt{S} \Uparrow_\Gamma^{!} \mathtt{!T}} \qquad \text{(Promote-OChan)}$$

$$\frac{\mathtt{S} \Uparrow_\Gamma \mathtt{/T} \quad \text{or} \quad \mathtt{S} \Uparrow_\Gamma \mathtt{/T}}{\mathtt{S} \Uparrow_\Gamma^{!} \mathtt{!T}} \qquad \text{(Promote-RChan)}$$

10.2 Judgement Forms

$\Gamma \vdash_r \mathtt{e}\ \mathit{ok}$	a typing for process `e` can be reconstructed under assumptions Γ
$\Gamma \vdash_r \mathtt{d} \rhd \Delta$	a typing for declaration `d` can be reconstructed yielding bindings Δ
$\Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{T}$	value (or path) `v` has type `T` (checking)
$\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{T}$	value (or path) `v` has type `T` (synthesis)
$\Gamma \vdash_r \mathtt{fv} \overleftarrow{\in} \mathtt{FT} \rhd \sigma$	field value `fv` has field type `FT` and yields substitution σ (checking)
$\Gamma \vdash_r \mathtt{fv} \overrightarrow{\in} \mathtt{FT} \rhd \sigma$	field value `fv` has field type `FT` and yields substitution σ (synthesis)
$\Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{T} \rhd \Delta$	pattern `p` requires type `T` and yields bindings Δ (checking)
$\Gamma \vdash_r \mathtt{p} \overrightarrow{\in} \mathtt{T} \rhd \Delta$	pattern `p` requires type `T` and yields bindings Δ (synthesis)
$\Gamma \vdash_r \mathtt{fp} \overleftarrow{\in} \mathtt{FT} \rhd \Delta$	field pattern `p` requires field type `FT` and yields bindings Δ (checking)
$\Gamma \vdash_r \mathtt{fp} \overrightarrow{\in} \mathtt{FT} \rhd \Delta$	field pattern `p` requires field type `FT` and yields bindings Δ (synthesis)
$\Gamma \vdash_r \mathtt{a} \overleftarrow{\in} \mathtt{T}$	abstraction `v` accepts type `T` (checking)
$\Gamma \vdash_r \mathtt{a} \overrightarrow{\in} \mathtt{T}$	abstraction `v` accepts type `T` (synthesis)
$\Gamma \vdash_r^p \mathtt{a} \overrightarrow{\in} \mathtt{T}$	abstraction `v` accepts type `T` (synthesis of pattern only)

$\Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overleftarrow{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma$
list of field values matches list of field types and yields substitution σ (checking)

The type reconstruction process yields a fully typed term, which is typeable according to the typechecking rules in the following chapter. So, strictly speaking, we should write $\Gamma \vdash_r \mathtt{e} \to \mathtt{e}'$ instead of $\Gamma \vdash_r \mathtt{e}\ ok$, and similarly for the other reconstruction judgements. However, since it is trivial to read off the required fully typed term from a type-reconstruction derivation, we leave it implicit to lighten the rules.

For some syntactic forms, the checking and synthesis rules are identical. In these cases, we write just one rule, using the notation $\overset{\leftrightarrow}{\in}$ to mean "either checking or synthesis." Where $\overset{\leftrightarrow}{\in}$ appears more than once in a rule, all occurrences are understood as pointing uniformly either right or left.

10.3 Processes

The reconstruction rules for the null process, for parallel composition, and for processes preceded by declarations are just like the corresponding typechecking rules.

$$\Gamma \vdash_r \mathtt{()}\ ok \qquad \text{(RE-NULL)}$$

$$\frac{\Gamma \vdash_r \mathtt{e}_1\ ok \qquad \Gamma \vdash_r \mathtt{e}_2\ ok}{\Gamma \vdash_r \mathtt{(e_1\ \mid\ e_2)}\ ok} \qquad \text{(RE-PRL)}$$

$$\frac{\Gamma \vdash_r \mathtt{d} \triangleright \Delta \qquad \Delta \vdash_r \mathtt{e}\ ok}{\Gamma \vdash_r \mathtt{(d\ e)}\ ok} \qquad \text{(RE-DEC)}$$

In an output expression, we synthesize the type of the channel, verify that it can be promoted to an output channel type, and then check that the type of the argument matches the type carried by the channel.

$$\frac{\Gamma \vdash_r \mathtt{v}_1 \overset{\rightarrow}{\in} \mathtt{T}_1 \Uparrow^{!}_{\Gamma} \mathtt{!T}_2 \qquad \Gamma \vdash_r \mathtt{v}_2 \overset{\leftarrow}{\in} \mathtt{T}_2}{\Gamma \vdash_r \mathtt{v_1!v_2}\ ok} \qquad \text{(RE-OUT)}$$

Similarly, in an input expression, we synthesize the type of the channel, promote it to an input channel type, and then check that the abstraction can accept a value of the type carried by the channel.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\rightarrow}{\in} \mathtt{T}_1 \Uparrow^{?}_{\Gamma} \mathtt{?T}_2 \qquad \Gamma \vdash_r \mathtt{a} \overset{\leftarrow}{\in} \mathtt{T}_2}{\Gamma \vdash_r \mathtt{v?a}\ ok} \qquad \text{(RE-IN)}$$

In a conditional expression, we simply check that the guard value has type `Bool` and verify that both branches are well formed.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\leftarrow}{\in} \mathtt{Bool} \qquad \Gamma \vdash_r \mathtt{e}_1\ ok \qquad \Gamma \vdash_r \mathtt{e}_2\ ok}{\Gamma \vdash_r \mathtt{if\ v\ then\ e_1\ else\ e_2}\ ok} \qquad \text{(RE-IF)}$$

10.4 Declarations

The type reconstruction rules for `new` and `run` have the same form as the corresponding typechecking rules.

$$\frac{\mathtt{T} =_{\Gamma} \mathtt{\hat{}U}}{\Gamma \vdash_r \mathtt{new\ x{:}T} \triangleright \Gamma, \mathtt{x{:}T}} \qquad \text{(RD-NEW)}$$

$$\frac{\Gamma \vdash_r \mathtt{e}\ ok}{\Gamma \vdash_r \mathtt{run\ e} \triangleright \Gamma} \qquad \text{(RD-RUN)}$$

To verify the consistency of a collection of recursive definitions, we first synthesize the types expected by each of the abstractions, ignoring the bodies of the abstractions (we cannot check these yet because we do not know the types of recursive occurrences of the bound variables yet); this is accomplished with the special "pattern synthesis only" judgement $\Gamma \vdash_r^p \mathtt{a}_i \overrightarrow{\in} \mathtt{T}_i$, which looks inside an abstraction and synthesizes the type of the pattern at the head. Now, knowing what types are expected, we check the types of the abstractions in the usual way in a typing context extended with an appropriate collection of responsive channel bindings; these bindings are also the result of the `def` form.

$$\frac{\begin{array}{c}\Gamma \vdash_r^p \mathtt{a}_i \overrightarrow{\in} \mathtt{T}_i \qquad \text{for each } i\\ \Gamma, \mathtt{x}_1\mathtt{:/T}_1, \ldots, \mathtt{x}_n\mathtt{:/T}_n \vdash_r \mathtt{a}_i \overleftarrow{\in} \mathtt{T}_i \qquad \text{for each } i\end{array}}{\Gamma \vdash_r \mathtt{def\ x_1a_1\ \ldots\ and\ x_na_n} \rhd \Gamma, \mathtt{x}_1\mathtt{:/T}_1, \ldots, \mathtt{x}_n\mathtt{:/T}_n} \qquad \text{(RD-DEF)}$$

We provide two reconstruction rules for `val` forms: one where the pattern on the left is explicitly typed and determines the type of the value on the right, and one where the value determines the type of the pattern. Note that these two cases may overlap, when both `p` and `v` can be synthesized. But (we conjecture) in this case the resulting fully typed term is the same regardless of which rule is applied. In practice the type reconstruction algorithm applies RD-VAL-1 first, attempting to synthesize the type of `p` and trying RD-VAL-2 if `p` does not contain enough annotations.

$$\frac{\Gamma \vdash_r \mathtt{p} \overrightarrow{\in} \mathtt{T} \rhd \Delta \qquad \Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{val\ p=v} \rhd \Delta} \qquad \text{(RD-VAL-1)}$$

$$\frac{\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{T} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{T} \rhd \Delta}{\Gamma \vdash_r \mathtt{val\ p=v} \rhd \Delta} \qquad \text{(RD-VAL-2)}$$

10.5 Abstractions

10.5.1 Process abstractions

For simple process abstractions, the checking and synthesis rules are the same: we either check or synthesize the type of the pattern, as appropriate, and then verify that the body is well formed.

$$\frac{\Gamma \vdash_r \mathtt{p} \overleftrightarrow{\in} \mathtt{T} \rhd \Delta \qquad \Delta \vdash_r \mathtt{e}\ ok}{\Gamma \vdash_r \mathtt{p\ =\ e} \overleftrightarrow{\in} \mathtt{T}} \qquad \text{(RACS-ABS)}$$

10.5.2 Synthesis of pattern types only

These rules are only called from the rule (RD-DEF) for mutually recursive definitions, where we must know the types of all the patterns before we can check the types of any of the bodies.

$$\frac{\Gamma \vdash_r \mathtt{p} \overrightarrow{\in} \mathtt{T} \rhd \Delta}{\Gamma \vdash_r^p \mathtt{p=e} \overrightarrow{\in} \mathtt{T}} \qquad \text{(RSPO-ABS)}$$

10.6 Patterns

Note the dual role of subtyping in the pattern rules given in this section and the value rules that follow.

10.6.1 Variable patterns

To check that a pattern consisting of an explicitly typed variable can accept a value of type `T`, we check that the explicit annotation `S` is a supertype of `T`. In the binding generated by the pattern, though, we give `x` only the (less informative) explicitly specified type `S`.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash_r \mathtt{x\!:\!S} \overleftarrow{\in} \mathtt{T} \rhd \Gamma, \mathtt{x\!:\!S}} \tag{RPC-Var-1}$$

Checking a variable pattern with no type annotation is trivial: the expected type determines the type of the binding generated by the pattern.

$$\Gamma \vdash_r \mathtt{x} \overleftarrow{\in} \mathtt{T} \rhd \Gamma, \mathtt{x\!:\!T} \tag{RPC-Var-2}$$

To synthesize a type for a variable pattern, an explicit type annotation must be present.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash_r \mathtt{x\!:\!T} \overrightarrow{\in} \mathtt{T} \rhd \Gamma, \mathtt{x\!:\!T}} \tag{RPS-Var}$$

10.6.2 Record patterns

To check that a record pattern has a type T, we first check that T can be promoted to a record type and then check one by one that the field patterns have the corresponding field types. The bindings generated by the record pattern include those generated by all the field patterns.

$$\frac{\mathtt{T} \Uparrow_\Gamma \mathtt{[l_1FT_1 \ldots l_nFT_n]} \qquad \Gamma \vdash_r \mathtt{fp_1} \overleftarrow{\in} \mathtt{FT_1} \rhd \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash_r \mathtt{fp_n} \overleftarrow{\in} \mathtt{FT_n} \rhd \Delta_n}{\Gamma \vdash_r \mathtt{[l_1fp_1 \ldots l_nfp_n]} \overleftarrow{\in} \mathtt{T} \rhd \Delta_n} \tag{RPC-Record}$$

To synthesize a record pattern type, we synthesize types for the field patterns and put them together in a record type.

$$\frac{\Gamma \vdash_r \mathtt{fp_1} \overrightarrow{\in} \mathtt{FT_1} \rhd \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash_r \mathtt{fp_n} \overrightarrow{\in} \mathtt{FT_n} \rhd \Delta_n}{\Gamma \vdash_r \mathtt{[l_1fp_1 \ldots l_nfp_n]} \overrightarrow{\in} \mathtt{[l_1FT_1 \ldots l_nFT_n]} \rhd \Delta_n} \tag{RPS-Record}$$

10.6.3 Rectype patterns

To check that a `rec` pattern accepts a recursive type T, we unroll T once to U and check that the body pattern p accepts U.

$$\frac{\Gamma \vdash \mathtt{T} \leadsto \mathtt{U} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{U} \rhd \Delta}{\Gamma \vdash_r \mathtt{(rec\ p)} \overleftarrow{\in} \mathtt{T} \rhd \Delta} \tag{RPC-Rec-1}$$

When an explicit type annotation S is given on a `rec` pattern, we check that S is a supertype of the expected type T and then unroll S to find the type U against which the body of the pattern is checked.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S} \qquad \Gamma \vdash \mathtt{S} \leadsto \mathtt{U} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{U} \rhd \Delta}{\Gamma \vdash_r \mathtt{(rec\!:\!S\ p)} \overleftarrow{\in} \mathtt{T} \rhd \Delta} \tag{RPC-Rec-2}$$

To synthesize a type for a `rec` pattern, an explicit type annotation must be present.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} \leadsto \mathtt{U} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{U} \rhd \Delta}{\Gamma \vdash_r \mathtt{(rec\!:\!T\ p)} \overrightarrow{\in} \mathtt{T} \rhd \Delta} \tag{RPS-Rec}$$

10.6.4 Wildcard patterns

A wildcard pattern without explicit type annotation has any desired type.

$$\Gamma \vdash_r \mathtt{\_} \overleftarrow{\in} \mathtt{T} \triangleright \Gamma \tag{RPC-Wild-1}$$

When an explicit annotation is present, we verify that it is a supertype of the expected type `T`.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash_r \mathtt{\_:S} \overleftarrow{\in} \mathtt{T} \triangleright \Gamma} \tag{RPC-Wild-2}$$

In a synthesis context, an explicit type annotation must be given.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash_r \mathtt{\_:T} \overrightarrow{\in} \mathtt{T} \triangleright \Gamma} \tag{RPS-Wild}$$

10.6.5 Layered patterns

To check that a layered pattern has a type `T`, we check that its body has type `T` and add a binding for the variable `x`, also with type `T`.

$$\frac{\Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{T} \triangleright \Delta}{\Gamma \vdash_r \mathtt{x@p} \overleftarrow{\in} \mathtt{T} \triangleright \Delta, \mathtt{x:T}} \tag{RPC-As-1}$$

When an explicit type annotation `S` is provided, we check that it is a supertype of `T` and then continue checking that the body accepts type `S`.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{S} \triangleright \Delta}{\Gamma \vdash_r \mathtt{x:S@p} \overleftarrow{\in} \mathtt{T} \triangleright \Delta, \mathtt{x:S}} \tag{RPC-As-2}$$

To synthesize a type for a layered pattern without explicit annotation, we continue synthesizing the type of the body.

$$\frac{\Gamma \vdash_r \mathtt{p} \overrightarrow{\in} \mathtt{T} \triangleright \Delta}{\Gamma \vdash_r \mathtt{x@p} \overrightarrow{\in} \mathtt{T} \triangleright \Delta, \mathtt{x:T}} \tag{RPS-As-1}$$

When an explicit annotation is provided, we enter checking mode for the body.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash_r \mathtt{p} \overleftarrow{\in} \mathtt{T} \triangleright \Delta}{\Gamma \vdash_r \mathtt{x:T@p} \overrightarrow{\in} \mathtt{T} \triangleright \Delta, \mathtt{x:T}} \tag{RPS-As-2}$$

10.7 Field Patterns

Patterns for value fields are just ordinary patterns: they are handled by the other pattern rules, as appropriate. Type fields are handled by the following rules.

When the expected field type includes a subtyping constraint `X<T` and the field pattern is also a subtyping constraint `X<S`, we check that `S` is more permissive that `T` and yield the binding `X<S` corresponding to the programmer's annotation.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash_r \mathtt{\#X<S} \overleftarrow{\in} \mathtt{\#X<T} \triangleright \Gamma, \mathtt{X<S}} \tag{RFPC-Type-1}$$

Similarly, when both the field pattern and the expected field type have equality constraints, we check that the two types are indeed equal.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \mathtt{S} =_\Gamma \mathtt{T}}{\Gamma \vdash_r \mathtt{\#X{=}S} \overleftarrow{\in} \mathtt{\#X{=}T} \rhd \Gamma, \mathtt{X{=}S}} \tag{RFPC-Type-2}$$

When the expected field type has an equality constraint `X=T` but the field pattern has a subtyping constraint `X<S`, we check that `S` is more permissive than `T` and return the less informative binding `X<S`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash_r \mathtt{\#X{<}S} \overleftarrow{\in} \mathtt{\#X{=}T} \rhd \Gamma, \mathtt{X{<}S}} \tag{RFPC-Type-3}$$

To synthesize a field type for a field pattern, we simply yield the stated field type and the corresponding binding.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash_r \mathtt{\#X{<}T} \overrightarrow{\in} \mathtt{\#X{<}T} \rhd \Gamma, \mathtt{X{<}T}} \tag{RFPS-Type-1}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash_r \mathtt{\#X{=}T} \overrightarrow{\in} \mathtt{\#X{=}T} \rhd \Gamma, \mathtt{X{=}T}} \tag{RFPS-Type-2}$$

10.8 Paths

10.8.1 Variables

To check that a variable has type `T`, we verify that its declared type `S` is a subtype of `T`.

$$\frac{\Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma_1, \mathtt{x}:\mathtt{S}, \Gamma_2 \vdash_r \mathtt{x} \overleftarrow{\in} \mathtt{T}} \tag{RPathC-Var}$$

We synthesize a type for a variable by looking it up in the context.

$$\Gamma_1, \mathtt{x}:\mathtt{T}, \Gamma_2 \vdash_r \mathtt{x} \overrightarrow{\in} \mathtt{T} \tag{RPathS-Var}$$

10.8.2 Field projection

To check that a field projection has type `T`, we synthesize the type of the record from which the projection is being made, obtain the appropriate field type $\mathtt{T}_i$, and check that it is a subtype of `T`.

$$\frac{\Gamma \vdash \mathtt{path} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma [\mathtt{l}_1\mathtt{T}_1 \ldots \mathtt{x}_i{=}\mathtt{T}_i \ldots \mathtt{l}_n\mathtt{T}_n] \qquad \Gamma \vdash \mathtt{T}_i < \mathtt{T}}{\Gamma \vdash \mathtt{path.x}_i \overleftarrow{\in} \mathtt{T}} \tag{RPathC-Label}$$

In synthesis mode, we just synthesize the type of the field and return it.

$$\frac{\Gamma \vdash \mathtt{path} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma [\mathtt{l}_1\mathtt{T}_1 \ldots \mathtt{x}_i{=}\mathtt{T}_i \ldots \mathtt{l}_n\mathtt{T}_n]}{\Gamma \vdash \mathtt{path.x}_i \overrightarrow{\in} \mathtt{T}_i} \tag{RPathS-Label}$$

10.9 Values

10.9.1 Checking Against `Top`

Since `Top` is the maximal type, checking against it is trivial: a value that has any type whatsoever has a type that is a subtype of `Top`.

$$\frac{\Gamma \vdash \mathtt{v} \overset{\rightarrow}{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{v} \overset{\leftarrow}{\in} \mathtt{Top}} \qquad \text{(RVC-Top)}$$

10.9.2 Constants

To check that a constant `k` has type `T`, we verify that `T` is a supertype of the constant's given type.

$$\frac{\Gamma \vdash \mathit{TypeOfConst}(\mathtt{k}) < \mathtt{T}}{\Gamma \vdash_r \mathtt{k} \overset{\leftarrow}{\in} \mathtt{T}} \qquad \text{(RVC-Const)}$$

In a synthesis context, the type of a constant is just its given type.

$$\Gamma \vdash_r \mathtt{k} \overset{\rightarrow}{\in} \mathit{TypeOfConst}(\mathtt{k}) \qquad \text{(RVS-Const)}$$

10.9.3 Local Declarations

In either checking or synthesis mode, the type of a value preceded by a declaration is found by calculating the bindings associated with the declaration and then checking or synthesizing the type of the body value, as appropriate.

$$\frac{\Gamma \vdash_r \mathtt{d} \rhd \Delta \qquad \Delta \vdash_r \mathtt{v} \overset{\leftrightarrow}{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{(d\ v)} \overset{\leftrightarrow}{\in} \mathtt{T}} \qquad \text{(RVCS-Dec)}$$

10.9.4 Conditionals

To check that a conditional value has type `T`, we first check that the guard has type `Bool`, then check that the branches both have type `T`.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\leftarrow}{\in} \mathtt{Bool} \qquad \Gamma \vdash_r \mathtt{v}_1 \overset{\leftarrow}{\in} \mathtt{T} \qquad \Gamma \vdash_r \mathtt{v}_2 \overset{\leftarrow}{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{if\ v\ then\ v_1\ else\ v_2} \overset{\leftarrow}{\in} \mathtt{T}} \qquad \text{(RVC-IfV-1)}$$

When an explicit result type `S` is provided, we check the branches against `S` and verify that `S` is a subtype of the expected type `T`.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\leftarrow}{\in} \mathtt{Bool} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash_r \mathtt{v}_1 \overset{\leftarrow}{\in} \mathtt{S} \qquad \Gamma \vdash_r \mathtt{v}_2 \overset{\leftarrow}{\in} \mathtt{S} \qquad \Gamma \vdash \mathtt{S} < \mathtt{T}}{\Gamma \vdash_r \mathtt{if{:}S\ v\ then\ v_1\ else\ v_2} \overset{\leftarrow}{\in} \mathtt{T}} \qquad \text{(RVC-IfV-2)}$$

To synthesize the type of a conditional value with no explicit annotation, we synthesize the type of the first branch and then check that the type of the second branch is a subtype.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\leftarrow}{\in} \mathtt{Bool} \qquad \Gamma \vdash_r \mathtt{v}_1 \overset{\rightarrow}{\in} \mathtt{T} \qquad \Gamma \vdash_r \mathtt{v}_2 \overset{\leftarrow}{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{if\ v\ then\ v_1\ else\ v_2} \overset{\rightarrow}{\in} \mathtt{T}} \qquad \text{(RVS-IfV-1)}$$

If an explicit result type is provided in a synthesis context, we check that the branches have this type and return it.

$$\frac{\Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{Bool} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash_r \mathtt{v}_1 \overleftarrow{\in} \mathtt{S} \qquad \Gamma \vdash_r \mathtt{v}_2 \overleftarrow{\in} \mathtt{S}}{\Gamma \vdash_r \mathtt{if{:}S\ v\ then\ v}_1\ \mathtt{else\ v}_2 \overrightarrow{\in} \mathtt{S}} \qquad \text{(RVS-IfV-2)}$$

10.9.5 Anonymous abstractions

To check that an anonymous abstraction `\a` has type `T`, we first convert `T` to a type of the form `/S` or `!S` and then check that `a` can accept type `S`.

$$\frac{\mathtt{T} =_\Gamma \mathtt{/S} \qquad \Gamma \vdash_r \mathtt{a} \overleftarrow{\in} \mathtt{S}}{\Gamma \vdash_r \mathtt{\backslash a} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RAC-AnonAbs-1)}$$

$$\frac{\mathtt{T} =_\Gamma \mathtt{!S} \qquad \Gamma \vdash_r \mathtt{a} \overleftarrow{\in} \mathtt{S}}{\Gamma \vdash_r \mathtt{\backslash a} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RAC-AnonAbs-2)}$$

To synthesize the type of an anonymous abstraction `\a`, we synthesize the type accepted by `a` and return a responsive channel type accepting elements of this type.

$$\frac{\Gamma \vdash_r \mathtt{a} \overrightarrow{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{\backslash a} \overrightarrow{\in} \mathtt{/T}} \qquad \text{(RAS-AnonAbs)}$$

10.9.6 Rectype values

To check that a `rec` value has type `T`, we first check that its explicitly given type `S` is a subtype of `T`, then unroll `S` once to obtain `U` and check that the body `v` has type `U`.

$$\frac{\Gamma \vdash \mathtt{S} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{S} < \mathtt{T} \qquad \Gamma \vdash \mathtt{S} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{U}}{\Gamma \vdash_r \mathtt{(rec{:}S\ v)} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RVC-Rec-1)}$$

When no explicit annotation is given, we unroll the expected recursive type `T` to obtain the expected type of the body.

$$\frac{\Gamma \vdash \mathtt{T} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{U}}{\Gamma \vdash_r \mathtt{(rec\ v)} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RVC-Rec-2)}$$

In a synthesis context, the explicit type annotation is required.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{T} \rightsquigarrow \mathtt{U} \qquad \Gamma \vdash_r \mathtt{v} \overleftarrow{\in} \mathtt{U}}{\Gamma \vdash_r \mathtt{(rec{:}T\ v)} \overrightarrow{\in} \mathtt{T}} \qquad \text{(RVS-Rec)}$$

10.9.7 Records

To check that a record has a type `T`, we convert `T` to the form of a record type, then check the fields of this type against the field values provided. A separate "field list typing" judgement form is used to check all of the fields together; this judgement takes care of "guessing" any type fields that may have been omitted by the programmer. (The substitution returned by the field list typing judgement is only needed in typechecking application expressions; here we throw it away.)

$$\frac{\mathtt{T} =_\Gamma \mathtt{[l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n\mathtt{]} \qquad \Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overleftarrow{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma}{\Gamma \vdash_r \mathtt{[l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m\mathtt{]} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RVC-Record)}$$

To synthesize the type of a record, we synthesize types for the fields one by one and put them together into a record type.

$$\frac{\Gamma \vdash_r \mathtt{fv}_1 \overrightarrow{\in} \mathtt{FT}_1 \triangleright \sigma_1 \quad \ldots \quad \Gamma \vdash_r \mathtt{fv}_n \overrightarrow{\in} \mathtt{FT}_n \triangleright \sigma_n}{\Gamma \vdash_r \mathtt{[l_1fv_1\ldots l_nfv_n]} \overrightarrow{\in} \mathtt{[l_1FT_1\ldots l_nFT_n]}} \qquad \text{(RVS-RECORD)}$$

10.9.8 With

For simplicity, we do not attempt to propagate information down when checking a `with`-value. Both checking and synthesis forms synthesize the types of the body `v` and the fresh fields and glue them together to form the resulting type. In the checking case, this type is compared with the expected type.

$$\frac{\begin{array}{c}\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma \mathtt{[l_1FT_1\ldots l_mFT_m]} \\ \Gamma \vdash_r \mathtt{fv}_{m+1} \overrightarrow{\in} \mathtt{FT}_{m+1} \triangleright \sigma_{m+1} \quad \ldots \quad \Gamma \vdash_r \mathtt{fv}_n \overrightarrow{\in} \mathtt{FT}_n \triangleright \sigma_n \\ \Gamma \vdash \mathtt{[l_1FT_1\ldots l_nFT_n]} < \mathtt{T} \qquad \mathit{distinct}(\mathtt{l_1 \ldots l_n})\end{array}}{\Gamma \vdash_r \mathtt{(v\ with\ l_{m+1}fv_{m+1}\ldots l_nfv_n)} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RVC-WITH)}$$

In the synthesis case, it is simply returned.

$$\frac{\begin{array}{c}\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma \mathtt{[l_1FT_1\ldots l_mFT_m]} \qquad \mathit{distinct}(\mathtt{l_1 \ldots l_n}) \\ \Gamma \vdash_r \mathtt{fv}_{m+1} \overrightarrow{\in} \mathtt{FT}_{m+1} \triangleright \sigma_{m+1} \quad \ldots \quad \Gamma \vdash_r \mathtt{fv}_n \overrightarrow{\in} \mathtt{FT}_n \triangleright \sigma_n\end{array}}{\Gamma \vdash_r \mathtt{(v\ with\ l_{m+1}fv_{m+1}\ldots l_nfv_n)} \overrightarrow{\in} \mathtt{[l_1FT_1\ldots l_nFT_n]}} \qquad \text{(RVS-WITH)}$$

10.9.9 Where

The rules for `where` clauses are straightforward: we synthesize the type of the body `v` and the types of the new fields. From these, we construct a new record type by replacing the types of overridden fields. Note that the field replcaement operation is only defined when the field labels $\mathtt{x}_1, \ldots, \mathtt{x}_m$ are a subset of the explicit field labels in `v`. In a checking context, this result type is compared with the expected type `T`.

$$\frac{\begin{array}{c}\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma \mathtt{[l_1FT_1\ldots l_nFT_n]} \\ \Gamma \vdash_r \mathtt{v}_1 \overrightarrow{\in} \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash_r \mathtt{v}_m \overrightarrow{\in} \mathtt{T}_m \\ \Gamma \vdash \mathtt{[\{x_1 \mapsto T_1\}\ldots\{x_m \mapsto T_m\}l_1FT_1\ldots l_nFT_n]} < \mathtt{T}\end{array}}{\Gamma \vdash_r \mathtt{(v\ where\ x_1=v_1\ldots x_m=v_m)} \overleftarrow{\in} \mathtt{T}} \qquad \text{(RVC-WHERE)}$$

In a synthesis context, it is simply returned.

$$\frac{\begin{array}{c}\Gamma \vdash_r \mathtt{v} \overrightarrow{\in} \mathtt{S} \Uparrow_\Gamma \mathtt{[l_1FT_1\ldots l_nFT_n]} \\ \Gamma \vdash_r \mathtt{v}_1 \overrightarrow{\in} \mathtt{T}_1 \quad \ldots \quad \Gamma \vdash_r \mathtt{v}_m \overrightarrow{\in} \mathtt{T}_m \\ \{\mathtt{l}'_1, \ldots, \mathtt{l}'_m\} \subseteq \{\mathtt{l}_1, \ldots, \mathtt{l}_n\}\end{array}}{\Gamma \vdash_r \mathtt{(v\ where\ x_1=v_1\ldots x_m=v_m)} \overleftarrow{\in} \mathtt{[\{x_1 \mapsto T_1\}\ldots\{x_m \mapsto T_m\}l_1FT_1\ldots l_nFT_n]}} \qquad \text{(RVS-WHERE)}$$

10.9.10 Application

To check that an application has type `T`, we first synthesize the type of the "function" `v` and promote it, if necessary, to an output channel type carrying a record. We then use the "field list typing" judgement (as in the record checking rule) to check that the arguments provided match the expected field types, filling in missing type fields and generating a substitution showing which concrete types were supplied for each of the type arguments. The type of the result is formed by applying this substitution to the polymorphic type `U` of the result channel. Finally, we check that the result type obtained in this way is a subtype of the expected type `T`.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\rightarrow}{\in} \mathtt{S} \Uparrow_{\Gamma}^{!} \mathtt{![l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n\ \mathtt{U]} \qquad \Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma \qquad \Gamma \vdash \sigma\mathtt{U} < \mathtt{/T}}{\Gamma \vdash_r \mathtt{(v\ l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m\mathtt{)} \overset{\leftarrow}{\in} \mathtt{T}} \quad \text{(RVC-App-1)}$$

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\rightarrow}{\in} \mathtt{S} \Uparrow_{\Gamma}^{!} \mathtt{![l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n\ \mathtt{U']} \qquad \Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma \qquad \Gamma \vdash \sigma\mathtt{U}' < \mathtt{U} < \mathtt{/T}}{\Gamma \vdash_r \mathtt{(:U\ v\ l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m\mathtt{)} \overset{\leftarrow}{\in} \mathtt{T}} \quad \text{(RVC-App-2)}$$

In synthesis mode, we obtain the result type $\sigma\mathtt{T}$ by promoting $\sigma\mathtt{U}$ to a responsive channel type.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\rightarrow}{\in} \mathtt{S} \Uparrow_{\Gamma}^{!} \mathtt{![l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n\ \mathtt{U]} \qquad \Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma \qquad \sigma\mathtt{U} \Uparrow_{\Gamma} \mathtt{/T}}{\Gamma \vdash_r \mathtt{(v\ l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m\mathtt{)} \overset{\rightarrow}{\in} \mathtt{T}} \quad \text{(RVS-App-1)}$$

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\rightarrow}{\in} \mathtt{S} \Uparrow_{\Gamma}^{!} \mathtt{![l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n\ \mathtt{U']} \qquad \Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}'_1\mathtt{FT}_1 \ldots \mathtt{l}'_n\mathtt{FT}_n \rhd \sigma \qquad \Gamma \vdash \sigma\mathtt{U}' < \mathtt{U} \qquad \mathtt{U} \Uparrow_{\Gamma} \mathtt{/T}}{\Gamma \vdash_r \mathtt{(:U\ v\ l}_1\mathtt{fv}_1 \ldots \mathtt{l}_m\mathtt{fv}_m\mathtt{)} \overset{\rightarrow}{\in} \mathtt{T}} \quad \text{(RVS-App-1)}$$

10.10 Field Values

10.10.1 Value Fields

For a value field consisting of value expressions, we use the ordinary rules for values to check or synthesize its type. We are expected to return a substitution recording any type bindings made by this field; there are none, so we return an empty substitution.

$$\frac{\Gamma \vdash_r \mathtt{v} \overset{\leftrightarrow}{\in} \mathtt{T}}{\Gamma \vdash_r \mathtt{v} \overset{\leftrightarrow}{\in} \mathtt{T} \rhd \{\}} \quad \text{(RFVCS-Value)}$$

10.10.2 Type fields

To check that a type field matches a given field type, we compare the given type with the expected one (using either subtyping or conversion, depending on whether the expected constraint uses subtyping or equality) and return a substitution recording the binding of the given type `T` to the variable `X`.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{T} < \mathtt{S}}{\Gamma \vdash_r \mathtt{\#T} \overset{\leftarrow}{\in} \mathtt{\#X<S} \rhd \{\mathtt{X} \mapsto \mathtt{T}\}} \quad \text{(RFVC-Bound)}$$

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \qquad \Gamma \vdash \mathtt{S} \in \mathtt{K} \qquad \mathtt{T} =_{\Gamma} \mathtt{S}}{\Gamma \vdash_r \mathtt{\#T} \overset{\leftarrow}{\in} \mathtt{\#X=S} \rhd \{\mathtt{X} \mapsto \mathtt{T}\}} \quad \text{(RFVC-Equal)}$$

In synthesis mode, we simply return a field type with an equality constraint, expressing full information about the field.

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K}}{\Gamma \vdash_r \mathtt{\#T} \overset{\rightarrow}{\in} \mathtt{\#X=T} \rhd \{\mathtt{X} \mapsto \mathtt{T}\}} \quad \text{(RFVS-Type)}$$

10.11 Field Lists

The typing of field lists is where reconstruction of omitted type arguments to polymorphic functions takes place. We are always given both a list of field values (with labels) and a list of expected field types (with labels). We walk down the lists field by field, always dealing with the leftmost field type and making a recursive call to take care of the rest.

The direction of the arrow superscript whether fields are checked or synthesized as they are reached. By default, all fields will be checked. But we also allow a type field appearing in the list of field types to be omitted from the list of values. In this case, we "guess" a value for the type field and continue with the rest of the fields, but switch to synthesis mode. (The reason for switching to synthesis mode is that the algorithm implemented in the compiler does not actually guess; instead, it proceeds and tries to determine the type later on, using the types of value fields further to the right. This means these value fields must be synthesized rather than checked, since the type we are trying to determine might appear in the types we would try to check them against.)

When we reach the point where the lists of field values and types are both empty, we stop and yield an empty substitution (onto which the substitutions corresponding to type fields are added as the stack of recursive calls returns).

$$\Gamma \vdash_r \langle \textit{empty FieldVal list} \rangle \overset{\leftrightarrow}{\in} \langle \textit{empty FieldType list} \rangle \rhd \{\} \qquad \text{(RFLCS-EMPTY)}$$

When the labels of the first field value and the first field type match, we use the usual field value checking or synthesis relation (as appropriate) to deal with it and make a recursive call to handle the rest of the fields.

$$\frac{\Gamma \vdash \mathtt{fv}_1 \overset{\leftarrow}{\in} \mathtt{FT}_1 \rhd \sigma_1 \qquad \Gamma \vdash_r \mathtt{l}'_2\mathtt{fv}_2 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma}{\Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1\ \mathtt{l}'_2\mathtt{fv}_2 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\leftarrow}{\in} \mathtt{l}_1\mathtt{FT}_1\ \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma_1, \sigma} \qquad \text{(RFLC-PRES-CHECK)}$$

$$\frac{\Gamma \vdash \mathtt{fv}_1 \overset{\rightarrow}{\in} \mathtt{FT}'_1 \rhd \sigma_1 \qquad \Gamma \vdash \mathtt{FT}'_1 < \mathtt{FT}_1 \qquad \Gamma \vdash_r \mathtt{l}'_2\mathtt{fv}_2 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\rightarrow}{\in} \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma}{\Gamma \vdash_r \mathtt{l}_1\mathtt{fv}_1\ \mathtt{l}'_2\mathtt{fv}_2 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\rightarrow}{\in} \mathtt{l}_1\mathtt{FT}_1\ \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma_1, \sigma} \qquad \text{(RFLC-PRES-SYNTH)}$$

If a type field is omitted, we guess a value for it, record this value in the substitution, and proceed, but we switch to synthesis mode for the remainder of the fields.

$$\frac{\text{either } \mathtt{l}_1 \neq \mathtt{l}'_1 \text{ or } \mathtt{l}_1 = \mathtt{l}'_1 = \langle \textit{empty} \rangle \text{ and } \mathtt{fv}_1 \text{ is a value field} \qquad \Gamma \vdash \mathtt{T}_1 \in \mathtt{Type} \qquad \Gamma \vdash \mathtt{S} < \mathtt{T}_1 \qquad \Gamma \vdash_r \mathtt{l}'_1\mathtt{fv}_1 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\rightarrow}{\in} \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma}{\Gamma \vdash_r \mathtt{l}'_1\mathtt{fv}_1 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\leftrightarrow}{\in} \mathtt{l}_1\mathtt{\#X{<}T}_1\ \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \{\mathtt{X} \mapsto \mathtt{S}\}, \sigma} \qquad \text{(RFLC-ABS-BOUND)}$$

When a type field with an equality constraint is omitted, we can fill it in uniquely. This means we can stay in checking mode if we started in checking mode.

$$\frac{\text{either } \mathtt{l}_1 \neq \mathtt{l}'_1 \text{ or } \mathtt{l}_1 = \mathtt{l}'_1 = \langle \textit{empty} \rangle \text{ or } \mathtt{fv}_1 \text{ is a value field} \qquad \Gamma \vdash_r \mathtt{l}'_1\mathtt{fv}_1 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\leftrightarrow}{\in} \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \sigma}{\Gamma \vdash_r \mathtt{l}'_1\mathtt{fv}_1 \ldots \mathtt{l}'_m\mathtt{fv}_m \overset{\leftrightarrow}{\in} \mathtt{l}_1\mathtt{\#X{=}T}_1\ \mathtt{l}_2\mathtt{FT}_2 \ldots \mathtt{l}_n\mathtt{FT}_n \rhd \{\mathtt{X} \mapsto \mathtt{T}_1\}, \sigma} \qquad \text{(RFLC-ABS-EQUAL)}$$

Chapter 11

Type-Directed Derived Forms

Now that we have built up the machinery needed to calculate types of values, we can get rid of some more high-level syntactic forms by translating them into lower-level forms.

11.1 Records

To translate a `with` expression, we use a `val` declaration to bind all of its fields, then explicitly construct a new record containing these fields plus the new ones. Here, though, the original record may contain type fields as well as value fields, and we need to use field patterns and corresponding field values of different forms depending on the forms of the fields. We use an extra `val` binding with an explicit type declaration to make sure that the type of the new record is as abstract as that of the original.

$$\frac{\begin{array}{l}\mathtt{S} =_\Gamma \mathtt{[l_1FT_1\ldots l_mFT_m]} \\ \Gamma \vdash \mathtt{(v{:}S\ with\ l_{m+1}fv_{m+1}\ldots l_nfv_n)} \in \mathtt{T}\end{array} \qquad (\mathtt{fp}_i,\mathtt{fv}_i) = \begin{cases} (\mathtt{y}_i\mathtt{:T}_i,\mathtt{y}_i) & \text{if } \mathtt{FT}_i = \mathtt{T}_i \\ (\mathtt{\#X{<}T}_i,\mathtt{\#X}) & \text{if } \mathtt{FT}_i = \mathtt{\#X{<}T}_i \\ (\mathtt{\#X{=}T}_i,\mathtt{\#X}) & \text{if } \mathtt{FT}_i = \mathtt{\#X{=}T}_i \end{cases}}{\begin{array}{c}\mathtt{(v{:}S\ with\ l_{m+1}fv_{m+1}\ldots l_nfv_n)} \Rightarrow \\ \mathtt{(val\ [l_1fp_1\ldots l_mfp_m]\ =\ v\ (val\ x{:}T\ =\ [l_1fv_1\ldots l_mfv_m\ l_{m+1}fv_{m+1}\ldots l_nfv_n]\ x))}\end{array}} \quad \text{(TR-WITH)}$$

Similarly, we translate a `where` expression by binding all the fields of the original record and building a new record where the new field value appears in the appropriate position.

$$\frac{\mathtt{S} =_\Gamma \mathtt{[y_1{=}T_1\ldots y_n{=}T_n]} \qquad \Gamma \vdash \mathtt{v_1} \in \mathtt{S_1} \quad \ldots \quad \Gamma \vdash \mathtt{v}_m \in \mathtt{S}_m}{\begin{array}{c}\mathtt{(v{:}S\ where\ x_1{=}v_1\ldots x_m{=}v_m)} \Rightarrow \\ \mathtt{(val\ [y_1{=}z_1{:}T_1\ldots y_n{=}z_n{:}T_n]\ =\ v\ (val\ w_1{:}S_1{=}v_1\ \ldots\ (val\ w_m{:}S_m{=}v_m} \\ \mathtt{[\{x_1 \mapsto w_1\}\ldots\{x_n \mapsto w_n\}y_1{=}z_1\ldots y_n{=}z_n]\quad)))}\end{array}} \quad \text{(TR-WHERE)}$$

Chapter 12

Derived Forms for CPS Conversion

The final phase of translation is a *continuation-passing* transform similar to those used in some compilers for functional languages (e.g. [App92]). In essence, we transform a complex value expression into a process that performs whatever computation is necessary and sends the final value along a designated *continuation channel*.

CPS-conversion is the only translation that is not completely local: in order to CPS-convert a complex value expression, we need to know the context in which its result will be used; it is this context that provides the continuation channel.

For convenience, we present CPS-conversion as a transformation function mapping a high-level process (possibly containing complex values anywhere within it) to a process in the core language all in one step. But the "recursive calls" of the transformation function for process expressions and abstractions do not use any information from the current context. Moreover, CPS-conversion is "essentially idempotent," in the sense that applying it to a process that is already in the core language yields a process expression that is behaviorally (though not textually) identical to the first. From these two facts, it follows that we can apply CPS-conversion freely to any process or abstraction subexpression in a program without changing the program's meaning, and that if we can reach a core-language program by applying CPS-conversion piecemeal to some of its subphrases, there is no need to CPS-convert the whole program at the end.

12.1 Judgement Forms

There are three judgement forms: one for CPS-converting a whole process expression, one for converting a value and arranging to send the result on a given continuation channel, and an auxiliary judgement for dealing with lists of record fields.

$[\![\mathtt{e}]\!] = \mathtt{e}'$	CPS-conversion transforms $\mathtt{e}$ to $\mathtt{e}'$
$[\![\mathtt{v} \rightarrow \mathtt{c}]\!] = \mathtt{e}$	$\mathtt{e}$ calculates value $\mathtt{v}$ and sends result to continuation $\mathtt{c}$
$[\![\mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n \rightarrow \mathtt{FT}_1 \ldots \mathtt{FT}_n]\!]\mathtt{c} = \mathtt{e}$	$\mathtt{e}$ calculates record value $[\mathtt{l}_1\mathtt{fv}_1 \ldots \mathtt{l}_n\mathtt{fv}_n]$ and sends result to $\mathtt{c}$

Many of the rules in this chapter introduce binders on their right-hand sides that do not appear on the left. The names of bound variables introduced in this way are assumed to be different from the names of any variables already occurring in the syntactic phrases replacing metavariables in rule instances.

12.2 Processes

To CPS-convert a parallel composition, we recursively convert its branches.

$$[\![(\mathtt{e}_1 \ \mid \ \mathtt{e}_2)]\!] = ([\![\mathtt{e}_1]\!] \ \mid \ [\![\mathtt{e}_2]\!]) \qquad \text{(Cps-Par)}$$

$$[\![()]\!] = () \qquad \text{(Cps-Null)}$$

To convert a conditional process, we convert its guard and both branches. The branches are processes, so they can be converted "in place." The guard, on the other hand, may be a complex value expression, while the evaluation rules for the core language (Chapter 13) only work if the guard is a constant boolean. We produce a conditional in this form by making up a fresh continuation channel `c` in the form of a definition that, when sent a value `x`, uses `x` as the guard of the conditional. In the scope of the declaration of `c`, we insert the process expression found by CPS-converting `v` with continuation `c`; this will be a process whose behavior is to calculate the boolean result of `v` and send it along `c`.

$$[\![\texttt{if v then } \mathtt{e}_1 \texttt{ else } \mathtt{e}_2]\!] = \texttt{(def c x:Bool = if x then } [\![\mathtt{e}_1]\!] \texttt{ else } [\![\mathtt{e}_2]\!] \quad [\![\mathtt{v} \to \mathtt{c}]\!]\texttt{)} \qquad \text{(CPS-IF)}$$

Similarly, to convert an input process `v?a`, we create a definition for the continuation `c`, whose body does the actual input, and start a process that calculates the channel that is the value of `v` and sends it to `c`. Note that we need to use the typing of `v` to construct the type annotation for the binding of `x`. (This, among other similar considerations, is why CPS-conversion happens after type reconstruction.)

$$\frac{\Gamma \vdash \mathtt{v} \in \mathtt{T}}{[\![\mathtt{v?a}]\!] = \texttt{(def c x:T = x?}[\![\mathtt{a}]\!] \quad [\![\mathtt{v} \to \mathtt{c}]\!]\texttt{)}} \qquad \text{(CPS-IN)}$$

An output process $\mathtt{v}_1\mathtt{!}\mathtt{v}_2$ is treated similarly, except that two (nested) continuations are needed, since both $\mathtt{v}_1$ and $\mathtt{v}_2$ can be complex values.

$$\frac{\Gamma \vdash \mathtt{v}_1 \in \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{v}_2 \in \mathtt{T}_2}{[\![\mathtt{v}_1\mathtt{!}\mathtt{v}_2]\!] = \texttt{(def } \mathtt{c}_1\ \mathtt{x}_1\mathtt{:T}_1 \texttt{ = (def } \mathtt{c}_2\ \mathtt{x}_2\mathtt{:T}_2 \texttt{ = } \mathtt{x}_1\mathtt{!}\mathtt{x}_2 \quad [\![\mathtt{v}_2 \to \mathtt{c}_2]\!]\texttt{)} \quad [\![\mathtt{v}_1 \to \mathtt{c}_1]\!]\texttt{)}} \qquad \text{(CPS-OUT)}$$

12.3 Processes with Declarations

While CPS-converting process expressions, we take the opportunity to desugar `val` and `run` declarations into simpler forms in the core language. (These transformations are not CPS-conversions, strictly speaking, but we incorporate them in the same translation function because they share the property of being non-local—e.g. it is not possible to translate a `run` declaration without knowing the process context in which it appears.)

A `run` declaration prefixing a process is translated to the parallel composition of the process with the body of the `run`.

$$[\![\texttt{(run } \mathtt{e}_1 \quad \mathtt{e}_2\texttt{)}]\!] = \texttt{(}[\![\mathtt{e}_1]\!] \texttt{ | } [\![\mathtt{e}_2]\!]\texttt{)} \qquad \text{(CPS-RUN)}$$

A value declaration `val p=v` with body `e` is translated to a definition `def c p =` $[\![\mathtt{e}]\!]$ (which, when sent a value, matches it against `p` and continues with `e`) plus a process that calculates `v` and sends it along `c`.

$$[\![\texttt{(val p = v} \quad \texttt{e)}]\!] = \texttt{(def c p = } [\![\mathtt{e}]\!] \quad [\![\mathtt{v} \to \mathtt{c}]\!]\texttt{)} \qquad \text{(CPS-VAL)}$$

The declaration forms `new` and `def` are in the core language; they are passed straight through the CPS transform.

$$[\![\texttt{(new x:T e)}]\!] = \texttt{(new x:T } [\![\mathtt{e}]\!]\texttt{)} \qquad \text{(CPS-NEW)}$$

$$[\![\texttt{(def } \mathtt{x}_1\mathtt{a}_1 \texttt{ ... and } \mathtt{x}_n\mathtt{a}_n \quad \texttt{e)}]\!] = \texttt{(def } \mathtt{x}_1[\![\mathtt{a}_1]\!] \texttt{ ... and } \mathtt{x}_n[\![\mathtt{a}_n]\!] \quad [\![\mathtt{e}]\!]\texttt{)} \qquad \text{(CPS-DEF)}$$

12.4 Abstractions

Since value-abstractions have already been dealt with (in Chapter 5), the only abstraction form we need to deal with here is simple process abstractions.

$$[\![\mathtt{p=e}]\!] = \mathtt{p=}[\![\mathtt{e}]\!] \qquad \text{(CPS-ABS)}$$

12.5 Values

To calculate a constant value and send its result along c, we just send it along c.

$$[\![\texttt{k} \to \texttt{c}]\!] = \texttt{c!k} \tag{Cps-Const}$$

Similarly, to calculate the value of a variable and send it along c, we just send it along c.

$$[\![\texttt{x} \to \texttt{c}]\!] = \texttt{c!x} \tag{Cps-Var}$$

To calculate the value of a conditional expression, we first calculate the value of the guard and send it along an internal continuation channel d. If the value received on d is true, we then start a process to calculate the value of $\texttt{v}_1$ and send it along c; otherwise we calculate the value of $\texttt{v}_2$ and send it along c.

$$\begin{array}{c}[\![\texttt{if:T v then v}_1\texttt{ else v}_2 \to \texttt{c}]\!] = \\ \texttt{(def d x:Bool = if x then } [\![\texttt{v}_1 \to \texttt{c}]\!] \texttt{ else } [\![\texttt{v}_2 \to \texttt{c}]\!]\ [\![\texttt{v} \to \texttt{d}]\!]\texttt{)}\end{array} \tag{Cps-IfV}$$

An anonymous abstraction \a is evaluated by making up a named definition with a fresh private name (and body a) and sending the name along the continuation channel.

$$[\![\texttt{\textbackslash a} \to \texttt{c}]\!] = \texttt{(def d } [\![\texttt{a}]\!] \texttt{ c!d)} \tag{Cps-AnonAbs}$$

A recursive value is translated by translating its body and wrapping the result in a rec before resending it on c.

$$\frac{\Gamma \vdash \texttt{T} \leadsto \texttt{U}}{\begin{array}{c}[\![\texttt{(rec:T v)} \to \texttt{c}]\!] = \\ \texttt{(def d x:U = c!(rec:T x) } [\![\texttt{v} \to \texttt{d}]\!]\texttt{)}\end{array}} \tag{Cps-Rec}$$

To translate a record value, we need to translate its fields and construct a new record value from the results. We do this using an auxiliary judgement, since the fields can be either values or types and only the former need to be translated. The auxiliary judgement recurses down the list of fields, copying type fields and transforming value fields; we give it the list of fields and set a marker (written $|$) at the beginning of the list of fields to indicate that none of them have been translated yet.

$$\frac{\Gamma \vdash \texttt{[l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n\texttt{]} \in \texttt{[l}_1\texttt{FT}_1\ldots\texttt{l}_n\texttt{FT}_n\texttt{]}}{[\![\texttt{[l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n\texttt{]} \to \texttt{c}]\!] = [\![\,|\,\texttt{l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n \to \texttt{c}]\!]} \tag{Cps-Record}$$

To translate the application expression $\texttt{(v l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n\texttt{)}$, we send the arguments $\texttt{fv}_1$ through $\texttt{fv}_n$ along the channel denoted by v, including the current continuation c as the final argument. (The result sent back by v is the result of the application expression, and so needs to be sent on c; we leave out the forwarding step and just tell v to send its result directly on c.)

$$[\![\texttt{(v l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n\texttt{)} \to \texttt{c}]\!] = [\![\texttt{v![l}_1\texttt{fv}_1\ldots\texttt{l}_n\texttt{fv}_n\texttt{ c]}]\!] \tag{Cps-App}$$

Finally, we have a set of rules for converting values preceded by declarations. These are essentially the same as the rules for processes with declarations.

$$[\![\texttt{(run e v)} \to \texttt{c}]\!] = \texttt{(}[\![\texttt{e}]\!]\texttt{ | }[\![\texttt{v} \to \texttt{c}]\!]\texttt{)} \tag{Cps-RunV}$$

$$[\![\texttt{(val p = v}_1\texttt{ v}_2\texttt{)} \to \texttt{c}]\!] = \texttt{(def d p = } [\![\texttt{v}_2 \to \texttt{c}]\!]\ [\![\texttt{v}_1 \to \texttt{d}]\!]\texttt{)} \tag{Cps-ValV}$$

$$[\![\texttt{(new x:T v)} \to \texttt{c}]\!] = \texttt{(new x:T } [\![\texttt{v} \to \texttt{c}]\!]\texttt{)} \tag{Cps-NewV}$$

$$[\![\texttt{(def x}_1\texttt{a}_1\texttt{ ... and x}_n\texttt{a}_n\texttt{ v)} \to \texttt{c}]\!] = \texttt{(def x}_1[\![\texttt{a}_1]\!]\texttt{ ... and x}_n[\![\texttt{a}_n]\!]\ [\![\texttt{v} \to \texttt{c}]\!]\texttt{)} \tag{Cps-DefV}$$

12.6 Field Lists

There are three rules for CPS-converting lists of field values, depending on whether (1) the marker $\mid$ is at the end of the list of fields, indicating that all fields have already been converted, (2) the field after the marker is a value field, which must be CPS-converted, or (3) the field after the marker is a type field, which is copied directly into the list of result fields. In case (2), the value field immediately following the marker is CPS-converted by creating a definition for a new channel $\mathtt{c}_i$ whose body is formed by recursively CPS-converting the remaining fields and starting a process to send the value for the ith field along $\mathtt{c}_i$.

$$[\![\mathtt{l}_1\mathtt{fv}_1\ldots\mathtt{l}_n\mathtt{fv}_n \mid \to \mathtt{c}]\!] = \mathtt{c!}[\mathtt{l}_1\mathtt{fv}_1\ldots\mathtt{l}_n\mathtt{fv}_n] \qquad \text{(CpsF-Done)}$$

$$\frac{\Gamma \vdash \mathtt{v}_i \in \mathtt{T}_i}{\begin{array}{c}[\![\mathtt{l}_1\mathtt{fv}_1\ldots \mid \mathtt{l}_i\mathtt{v}_i\ldots\mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!] = \\ \mathtt{(def\ c}_i\ \mathtt{x}_i\mathtt{:T}_i\ \mathtt{=}\ [\![\mathtt{l}_1\mathtt{fv}_1\ldots\mathtt{l}_i\mathtt{x}_i \mid \mathtt{l}_{i+1}\mathtt{fv}_{i+1}\ldots\mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!]\quad [\![\mathtt{v}_i \to \mathtt{c}_i]\!]\mathtt{)}\end{array}} \qquad \text{(CpsF-Value)}$$

$$\begin{array}{c}[\![\mathtt{l}_1\mathtt{fv}_1\ldots \mid \mathtt{l}_i\mathtt{\#T}_i\ldots\mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!] = \\ [\![\mathtt{l}_1\mathtt{fv}_1\ldots\mathtt{l}_i\mathtt{\#T}_i \mid \mathtt{l}_{i+1}\mathtt{fv}_{i+1}\ldots\mathtt{l}_n\mathtt{fv}_n \to \mathtt{c}]\!]\end{array} \qquad \text{(CpsF-Type)}$$

Chapter 13

Evaluation

At last, we are ready to define the operational semantics of the Pict core language. This is essentially the same as the familiar "chemical abstract machine style" semantics for the pi-calculus [Mil91, etc.], with a few technical differences:

1. We retain `def` declarations in the core language instead of desugaring them into replicated input processes. The main reason for this is to avoid complicating the type system: in the present formulation, definitions give rise directly to channels with responsive types (`/T`). If we transformed definitions to replicated inputs, we would need to refine the type system to be able to check that these inputs were indeed responsive (cf. [KPT96]).

   The reduction rules for communicating with definitions are reminiscent of a simple fragment of the *join-calculus* [FG96].

2. We retain conditional processes in the core language instead of desugaring them using a standard "Church boolean" encoding. This is purely a matter of technical convenience: adding reduction rules for conditionals costs nothing here in terms of complexity and permits us to maintain the abstractness of the primitive `Bool` type.

13.1 Judgement Forms

As usual, there are two judgement forms for evaluation: a *structural congruence* that allows processes to be rearranged so as to bring the parts of redices into proximity and a *reduction relation* that shows how subprocesses actually interact.

$$\begin{array}{ll} \mathtt{e} \equiv \mathtt{e}' & \mathtt{e} \text{ is structurally congruent to } \mathtt{e}' \\ \mathtt{e} \rightarrow \mathtt{e}' & \mathtt{e} \text{ reduces in one step to } \mathtt{e}' \end{array}$$

13.2 Auxiliary Definitions

13.2.1 Substitution

In order to support planned extensions of the language, we want to maintain the property that the head of every path is a variable, not an explicitly constructed record. To maintain this property during reduction, we need to make sure that, when we substitute a record value for a variable during a communication step, any projection expressions with this variable as their head are reduced at the same time by projecting out the appropriate field:

$$\{\mathtt{x} \mapsto \mathtt{[\ldots l}_i\mathtt{v}_i\mathtt{\ldots]}\}\mathtt{x.l}_i \quad = \quad \mathtt{v}_i$$

This refined version of substitution is the one used in the rules that follow. (This needs to happen recursively for all sequences of projections.)

13.2.2 Matching

When a communication occurs, we match the value provided by the sender against the pattern in the receiver to yield a substitution that is applied to the body of the receiver. This substitution is defined as follows (using an auxiliary definition of "record field matching" to calculate the substitutions arising from the fields of a record pattern and corresponding value).

$$
\begin{array}{rcl}
\{\mathtt{x:T} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \\
\{\mathtt{\_:T} \mapsto \mathtt{v}\} & = & \{\,\} \\
\{\mathtt{(x:T@p)} \mapsto \mathtt{v}\} & = & \{\mathtt{x} \mapsto \mathtt{v}\} \cup \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{(rec:T\ p)} \mapsto \mathtt{(rec:S\ v)}\} & = & \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{[l_1fp_1 \ldots l_nfp_n]} \mapsto \mathtt{[l_1fv_1 \ldots l_nfv_n \ldots]}\} & = & \{\mathtt{fp_1} \mapsto_f \mathtt{fv_1}\} \cup \cdots \cup \{\mathtt{fp_n} \mapsto_f \mathtt{fv_n}\}
\end{array}
$$

$$
\begin{array}{rcl}
\{\mathtt{p} \mapsto_f \mathtt{v}\} & = & \{\mathtt{p} \mapsto \mathtt{v}\} \\
\{\mathtt{\#X<S} \mapsto_f \mathtt{T}\} & = & \{\mathtt{X} \mapsto \mathtt{T}\} \\
\{\mathtt{\#X=S} \mapsto_f \mathtt{T}\} & = & \{\mathtt{X} \mapsto \mathtt{T}\}
\end{array}
$$

13.3 Structural Congruence

The structural congruence rules allow parallel compositions to be commuted and reassociated at will, and null elements to be introduced and deleted.

$$\mathtt{((e_1\ \mid\ e_2)\ \mid\ e_3)} \equiv \mathtt{(e_1\ \mid\ (e_2\ \mid\ e_3))} \qquad \text{(Str-Assoc)}$$

$$\mathtt{(e_1\ \mid\ e_2)} \equiv \mathtt{(e_2\ \mid\ e_1)} \qquad \text{(Str-Comm)}$$

$$\mathtt{(e\ \mid\ ())} \equiv \mathtt{e} \qquad \text{(Str-Null)}$$

The critical direction of the *scope extrusion* rule allows the scope of declarations to expand to include adjacent parallel components, provided that no free variables become bound in the process. (According to our conventions about uniqueness of binders, this condition will always be satisfied; we retain it and similar ones below only as reminders.)

$$\frac{BV(\mathtt{d}) \cap FV(\mathtt{e_2}) = \emptyset}{\mathtt{((d\ e_1)\ \mid\ e_2)} \equiv \mathtt{(d\ (e_1\ \mid\ e_2))}} \qquad \text{(Str-Extrude)}$$

We also allow the ordering of adjacent declarations to be reversed, as long as neither declaration depends on variables bound by the other.

$$\frac{BV(\mathtt{d_1}) \cap FV(\mathtt{d_2}) = \emptyset \qquad BV(\mathtt{d_2}) \cap FV(\mathtt{d_1}) = \emptyset}{\mathtt{(d_1\ (d_2\ e))} \equiv \mathtt{(d_2\ (d_1\ e))}} \qquad \text{(Str-SwapDec)}$$

Finally, we allow two adjacent definitions to be combined into a single one.

$$\frac{\begin{array}{c}(\{\mathtt{x_1},\ldots,\mathtt{x_m}\} \cap \{\mathtt{x_{m+1}},\ldots,\mathtt{x_n}\} = \emptyset \\ (FV(\mathtt{a_1}) \cup \cdots \cup FV(\mathtt{a_m})) \cap \{\mathtt{x_{m+1}},\ldots,\mathtt{x_n}\} = \emptyset\end{array}}{\begin{array}{c}\mathtt{(def\ x_1a_1\ \ldots\ and\ x_ma_m\ (def\ x_{m+1}a_{m+1}\ \ldots\ and\ x_na_n\ \ e))} \equiv \\ \mathtt{(def\ x_1a_1\ \ldots\ and\ x_ma_m\ and\ x_{m+1}a_{m+1}\ \ldots\ and\ x_na_n\ \ e)}\end{array}} \qquad \text{(Str-Coalesce)}$$

The full structural congruence relation $\equiv$ is the least congruence on process expressions closed under these rules.

13.4 Reduction

The first reduction rule states that the reduction relation is closed under structural congruence, allowing the processes on both sides to be rearranged so as to give the other rules a chance to apply.

$$\frac{\mathtt{e}_1 \equiv \mathtt{e}_2 \rightarrow \mathtt{e}_3 \equiv \mathtt{e}_4}{\mathtt{e}_1 \rightarrow \mathtt{e}_4} \qquad \text{(RED-STR)}$$

We allow reduction inside one branch of a parallel composition and under declarations.

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_3}{(\mathtt{e}_1 \mid \mathtt{e}_2) \rightarrow (\mathtt{e}_3 \mid \mathtt{e}_2)} \qquad \text{(RED-PRL)}$$

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_2}{(\mathtt{d}\ \mathtt{e}_1) \rightarrow (\mathtt{d}\ \mathtt{e}_2)} \qquad \text{(RED-DEC)}$$

When a sender and a receiver on the same channel are placed in parallel, they can communicate and become the appropriate substitution instance of the receiver's body.

$$\frac{\{\mathtt{p} \mapsto \mathtt{v}\} \text{ defined}}{(\mathtt{x!v} \mid \mathtt{x?p = e}) \rightarrow \{\mathtt{p} \mapsto \mathtt{v}\}(\mathtt{e})} \qquad \text{(RED-COMM)}$$

Similarly, an output process can communicate with the definition that binds the channel on which it is sending, yielding an appropriate substitution instance of the definition's body. (The extra process `e` in the parallel composition is used to collect whatever other processes may be in the scope of the same declaration.)

$$\frac{\{\mathtt{p}_i \mapsto \mathtt{v}\} \text{ defined}}{\begin{array}{c}(\mathtt{def}\ \mathtt{x}_1\mathtt{p}_1 = \mathtt{e}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{p}_n = \mathtt{e}_n\ \ (\mathtt{x}_i\mathtt{!v} \mid \mathtt{e}))\\ \rightarrow (\mathtt{def}\ \mathtt{x}_1\mathtt{p}_1 = \mathtt{e}_1\ \ldots\ \mathtt{and}\ \mathtt{x}_n\mathtt{p}_n = \mathtt{e}_n\ \ (\{\mathtt{p}_i \mapsto \mathtt{v}\}(\mathtt{e}_i) \mid \mathtt{e}))\end{array}} \qquad \text{(RED-DEF)}$$

Finally, we reduce a conditional process expression to the appropriate branch, depending on the value of the guard.

$$\mathtt{if\ true\ then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rightarrow \mathtt{e}_1 \qquad \text{(RED-IF-T)}$$

$$\mathtt{if\ false\ then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rightarrow \mathtt{e}_2 \qquad \text{(RED-IF-F)}$$

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118) and as DEC Systems Research Center Research Report number 62, August 1990.

[Ama94] Roberto M. Amadio. Translating core Facile. Technical Report ECRC-TR-3-94, European Computer-Industry Research Center, GmbH, Munich, 1994. Also available as a technical report from CRIN(CNRS)-Inria (Nancy).

[AP94] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. Technical Report ECRC-M2-R10, European Computer-Industry Research Center, GmbH, Munich, 1994.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ber93] Bernard Berthomieu. Programming with behaviours in an ML framework. the syntax and semantics of LCS. Technical Report 93133, LAAS-CNRS, April 1993.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *ACM Principles of Programming Languages*, January 1992.

[BS94] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. Technical Report ECS–LFCS–94–297, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Report DAIMI PB-208, Computer Science Department, University of Aarhus, Denmark, 1986.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A Symmetric Integration of Concurrent and Functional Programming. In *Theory and Practice of Software Development (TAPSOFT)*, pages 184–209. Springer, 1989. LNCS 352.

[Hol83] Sören Holmström. PFL: A functional language for parallel programming, and its implementation. Programming Methodology Group, Report 7, University of Goteborg and Chalmers University of Technology, September 1983.

[Jon93] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 158–172. Springer-Verlag, 1993.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Principles of Programming Languages*, 1996.

[Mat91] David Matthews. A distributed concurrent implementation of Standard ML. Technical Report ECS-LFCS-91-174, University of Edinburgh, August 1991.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathematical Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[Pie96] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1996.

[PS96] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1996. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[PT97a] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus, 1997. Compiler, documentation, demonstration programs, and standard libraries; available electronically.

[PT97b] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.

[PT97c] Benjamin C. Pierce and David N. Turner. Pict libraries manual. Available electronically, 1997.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[San92] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[San93] Davide Sangiorgi. An investigation into functions as processes. In *Proc. Ninth International Conference on the Mathematical Foundations of Programming Semantics (MFPS'93)*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer Verlag, 1993.

[San94a] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[San94b] Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. Technical Report ECS–LFCS–94–282, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994. An extract appeared in *Proc. TACS '94*, Lecture Notes in Computer Science 789, Springer Verlag.

[Wal94] David Walker. Algebraic proofs of properties of objects. In *Proceedings of European Symposium on Programming*. Springer-Verlag, 1994.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.