An implementation of $F_{<:}$

Luca Cardelli
Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto CA 94301

Abstract

F<: is a highly expressive typed λ-calculus with subtyping. This paper describes an implementation of *F<:* (extended with recursive types), and documents the algorithms used. Using this implementation, one can test *F<:* programs and examine typing derivations.

To facilitate the writing of complex *F<:* encodings, we provide a flexible syntax-extension mechanism. New syntax can be defined from scratch, and the existing syntax can be extended on the fly. It is possible to introduce new binding constructs, while avoiding problems with variable capture.

To reduce the syntactic clutter, we provide a practical type inference mechanism that is applicable to any explicitly typed polymorphic language. Syntax extension and type inference interact in useful ways.

SRC Research Report 97, February 23, 1993.
© Digital Equipment Corporation 1993.
This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individuals contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Contents

1. Introduction 3
2. Overview 3
3. Syntax extension 6
    3.1 Grammars 6
    3.2. Syntax 8
    3.3. Actions 10
    3.4. How it is done 11
    3.5. Related work on syntax extension 13
4. Mock-modules and save-points 14
5. Top-level phrases 15
6. Type inference by “argument synthesis” 17
7. Recursion 19
Appendices 21
    Appendix A. Examples 21
    Appendix B. Lexicon 28
    Appendix C. Syntax 29
    Appendix D. Typing rules 32
    Appendix E. Typing algorithm 34
    Appendix F. Typing algorithm with argument synthesis 38
    Appendix G. Recursion 44
Acknowledgments 50
References 51

1. Introduction

F<: is a typed λ-calculus with subtyping. It is intended to capture the essence of subtyping and, to some extent, of object-oriented programming [Cardelli, *et al.* 1991; Curien, Ghelli 1991]. The *F<:* calculus was designed to be as small as possible, so that it could be studied formally. Its small size also happens to facilitate implementation; during its construction it was possible to explore some advanced techniques that should be useful for larger languages.

This paper describes the F-sub program, which is an implementation of *F<:* . (We assume a superficial familiarity with the latter.) Using this program, one can typecheck and evaluate *F<:* expressions and definitions, and examine typing and subtyping derivations. In order to keep the critical typing code clean and correct, the implementation is very minimal and supports only the basic constructs of *F<:*. This minimality, while having some pragmatic disadvantages, allows us to describe the fundamental algorithms in full detail in terms of an operational semantics that is faithful to the actual program code.

The operational semantics is described in the Appendix in layers of increasing complexity, the final layer corresponding closely to actual program code. The first layer corresponds to the typechecking algorithm for pure *F<:*. Then, other features are added: (a) de Bruijn indices, (b) partial type inference for second-order types, and (c) a new technique for integrating recursive types with second-order polymorphic types.

Apart from the typing algorithms, another aspect of the implementations should be of general interest. The extensible syntax mechanism we have implemented should be useful in other mechanized formal systems that need to define mathematical notation on the fly, such as theorem provers, proof checkers, and symbolic algebra systems. In these systems, one wishes to minimize the number of constructs in order to keep the difficult core algorithms clean and manageable. In the case of F-sub, we wish to keep the typing code simple by not providing basic data structures and control structures, requiring instead that they be encoded as λ-terms. The drawback of this approach is that after a few levels of encoding even simple programs become quite unreadable. To improve readability of the encodings, the F-sub system supports a very flexible syntax-extension mechanism based on an LL(1) parser. One can define entirely new grammars, or enrich the existing F-sub grammar. In particular, one can define new binding constructs and their associated meaning, while avoiding problems with variable capture.

The F-sub system consists of about 10,000 lines of Modula-3 code [Nelson 1991], equally partitioned between a reusable parsing package and F-sub proper. The implementation is portable to any computer running Modula-3, that is to almost any computer running a standard C compiler [Kalsow, Muller FTP]. Program sources and binaries for standard architectures are freely available [Cardelli FTP].

2. Overview

The syntax of F-sub types and terms is given below, informally. As a general convention, term-related names begin with a lower-case letter, while type-related names begin with an upper-case letter.

`A,B ::=`	types
`X`	type variables
`Top`	the biggest type
`A->B`	function spaces

	`All(`*X*`<:`*A*`)`*B*	bounded quantification
	`{`*A*`}`	grouping
a`,`*b* `::=`		terms
	x	term variables
	`top`	the canonical member of `Top`
	`fun(`*x*`:`*A*`)`*b*	functions
	b`(`*a*`)`	applications
	`fun(`*X*`<:`*A*`)`*b*	polymorphic functions
	b`(:`*A*`)`	type applications
	`{`*a*`}`	grouping

When loaded, the F-sub system displays its ‘`- `’ prompt, at which one can write a term like ‘`top`’, followed by a semicolon. The system answers by inferring the type of the term and evaluating it. The answers given by the system are indicated by ‘⇨ ’.

```
- top;
⇨ top : Top
```

In general, at the ‘`- `’ prompt one can write a *phrase*, always terminated by a semicolon. There are several kinds of phrases. The one above is a *term phrase*, while the one shown below is a *type phrase*; this is always preceded by a colon and causes the evaluation of a type:

```
- :Top;
⇨ : Top
```

Type definition phrases, introduced by ‘`Let`’ and *term definition phrases*, introduced by ‘`let`’, allow one to bind types and terms to variables:

```
- Let Id = All(X) X->X
⇨ Let Id <: Top = <Id>

- :Id;
⇨ : <Id>

- let id : Id = fun(X) fun(x:X) x;
⇨ let id : <Id> = <id>

- id;
⇨ <id> : <Id>
```

The system produces some answers in angle brackets, as an abbreviation, to avoid printing excessive details. If a term or a type has been given a name in a definition, then that term or type is printed as its given name in angle brackets. This printing heuristic has no effect on typing or evaluation.

Once a function like ‘`id`’ is defined, it can be applied to types and terms. A type application has the form ‘*a*`(:`*A*`)`’ (note the ‘`:`’); a term application has the form ‘*a*`(`*b*`)`’.

```
- id(:Id);
⇨ {fun(x:<Id>)x} : {<Id>-><Id>}

- id(:Id)(id);
⇨ <id> : <Id>
```

```
- id(:Id->Id);
➪ {fun(x:<Id>-><Id>)x} : {{<Id>-><Id>}->{<Id>-><Id>}}
```

The evaluator does not perform reductions inside functions:

```
- fun(x:Id) id(:Id)(x);
➪ {fun(x:<Id>)<id>(:<Id>)(x)} : {<Id>-><Id>}
```

As you may notice from the printed output, curly brackets, instead of parentheses, are used to group syntax:

```
- {fun(x:Top)x}(top);
➪ top : Top
```

Programs can be stored in files. For example we can prepare a file called ‘`Test.fsub`’ containing the Church encoding of booleans:

```
Let Bool = All(X) X->X->X;

let true: Bool = fun(X) fun(x:X) fun(y:X) x
    false: Bool = fun(X) fun(x:X) fun(y:X) y;
```

We can then load this file into the system by a *load phrase*:

```
- load Test;
```

According to the encoding of booleans above, a conditional of the form ‘`if x then false else true end`’ is written as:

```
x(:Bool)(false)(true)
```

It is possible, however, to define a more familiar syntax for conditionals by a *syntax extension*, as follows.

A *syntax phrase* introduces a new grammar or, in this example, modifies the existing one:

```
- syntax
     termBase ::= ...
        [ "if" term_1 "then" term_2 "else" term_3
          "giving" type_4 "end" ]
        =>  _1(:_4)(_2)(_3);
```

To understand this example, one must first know that ‘`termBase`’, ‘`term`’, and ‘`type`’ are some of the syntactic categories of F-sub given in Appendix C (a ‘`termBase`’ is a ‘`term`’ except for the right-recursive syntax of applications). Here we wish to modify the syntax of ‘`termBase`’ by taking its existing definition (indicated by ‘`...`’) and adding conditional expressions. By this mechanism we truly modify the recursive definition of terms; meaning that conditional expressions can be nested.

The grammar of conditionals is given above as a *sequence* (in square brackets) of keywords and numbered ‘`term`’ and ‘`type`’ grammars. The numbers are used in the *action* part of the grammar (following ‘`=>`’), where the relevant pieces of the input are reassembled into the encoding of conditionals shown earlier.

With the extended grammar we can write, for example:

```
- let not =
    fun(x:Bool)
      if x then false else true giving Bool end;
⇨ let not : {<Bool>-><Bool>} = <not>
```

As another example of syntax extensions, we can define *let terms* (as opposed to the top-level-only '`let`' definitions), translating them into functions and applications:

```
- syntax
    termBase ::= ...
      [ "let" termIde_1 ":" type_2 "=" term_3
        "in" term_4 "end" ]
      =>  {fun(_1:_2)_4}(_3);
```

In this example we are creating a new *binding* construct. This is reflected by the use of '`fun(_1:`' in the action part. Here '`_1`' refers to a '`termIde`', which is the F-sub grammar for a term identifier. Note that '`_4`' is inside the scope of '`_1`', producing the expected variable capture. (Unwanted variable captures are carefully avoided.) To try this out, we need to wrap the let-expressions in brackets to avoid confusion with let-phrases:

```
- {let x: Bool = true in not(x) end};
⇨ <false> : <Bool>
```

In general, a *term action* (preceded by '=>') can be any F-sub term, possibly containing *pattern variables* '*_n*'. Similarly, a *type-action* (preceded by '`:>`') can be any F-sub type, possibly containing pattern variables. An action can be appended to any piece of grammar. The pattern variables '*_n*' can similarly be appended to any piece of grammar, using parentheses '`(`','`)`' for grouping if necessary. After the definition of a syntax extension for terms or types, the new syntax can be used in the action parts of later grammars.

As an exercise, one could now try to define the syntax of existential types '`Some(X<:A)B`', giving the translation into universal types '`All(Y){All(X<:A)B->Y}->Y`'. For more complex tasks one should first read section 3.3 on Actions. (Exercise hints. One has to modify '`typeBase`', and capture a '`typeIde`' and two '`type`'s. The symbol for type actions is '`:>`', not '=>'. To see what the parser produces, write '`do ShowParsing On`;'.)

3. Syntax extension

In this section we describe a notation for grammars and its use in defining syntax extensions. This notation is used also in Appendix C to describe the formal syntax of F-sub.

3.1 Grammars

Our meta-notation for grammars is slightly non-standard. Moreover, its meaning is tightly associated with a particular parser (recursive descent). The reason for these peculiarities is that the same notation is used also for the syntax-extension facility within the language.

Terminal symbols are called *tokens*; the most important kinds of tokens are identifiers '`ob1`', delimiters '`)`', and quoted strings '`"abc"`'. The identifiers can be either alphanumeric '`ob1`' or symbolic '`->`'.

Moreover, identifiers are split into keyword and non-keyword classes; keywords are not legal variable names in binding constructs. See Appendix B for the full lexical details.

A *grammar description* g is one of the following constructions:

x	An identifier x represents a non-terminal grammar symbol, which must be bound to a grammar description. Parsing x is the same as parsing the associated description.
ide	The constant *ide* denotes a non-keyword identifier token. Parsing this succeeds when the next input token is a non-keyword identifier.
`"fun"`	A non-empty quoted string denotes the keyword or delimiter token given in quotes. Parsing this succeeds when the next input token is the given keyword or delimiter.
string	The constant *string* denotes a quoted-string token. Parsing this succeeds when the next input token is a quoted string.
$[g_1 \dots g_n]$	Square brackets denote a *sequence* of grammars. Parsing this succeeds if parsing each g_i in sequence succeeds. Parsing *[]* always succeeds.
$\{g_1 \dots g_n\}$	Curly brackets denote a *choice* of grammars. Parsing this succeeds if parsing one of the g_i's succeeds when trying them left to right. Parsing *{}* always fails. (If one of the g_i's fails *after* successfully parsing an input token, then the entire parsing fails, but this can happen only if the grammar is not LL(1).)
$(g_1 * g_2)$	This *iteration* construct is equivalent to the grammar $[g_1\ x]$ where $x ::= \{[g_2\ x]\ []\}$. However, the parsing of $(g_1 * g_2)$ can build left-associative parse-trees (in conjunction with *actions*), which are not otherwise representable by a non-left-recursive grammar.

A complete *grammar* has the form:

$$x_1 ::= g_1 \ \dots \ x_n ::= g_n$$

where $n \geq 1$, the x_i are distinct, and any x occurring in one of the g_i is one of the x_i. Moreover, the grammar must be non-left-recursive and LL(1) (where 1 refers to one token, not one character). The grammar so defined is the one defined by x_1.

As an example, here is a non-ambiguous grammar for untyped λ-terms:

lambda ::= { ide func appl }
func ::= ["[" ide "." lambda "]"]
appl ::= ["(" lambda lambda ")"]

Suppose now we wish to change the syntax of application from ‘(a b)’ to ‘a(b)’. The grammar becomes left-recursive, but this problem can be eliminated by distinguishing between simple terms and complex terms as shown below. The resulting grammar is LL(1), and the recursive-descent parser resolves any ambiguity:

lambda ::= [simple arg]
arg ::= { [par arg] [] }
simple ::= { ide func par}
func ::= ["fun" ide "." lambda]
par ::= [“(" lambda ")"]

The grammar above parses λ-terms, but because of the way left-recursion was eliminated, application associates to the right (that is, a(b)(c) parse as <a<(b)(c)>> instead of <<a(b)>(c)>), which complicates further processing. This problem can be solved by the iteration operator '*', which intentionally associates to the left. The grammar of λ-terms should then be expressed as follows:

*lambda ::= (simple * par)*
simple ::= { ide func par}
func ::= ["fun" ide "." lambda]
par ::= ["(" lambda ")"]

Implementation-specific warning: when used in the syntax-extension facility, non-LL(1) grammars will typically cause parsing failures, and left-recursive grammars will cause non-termination. This is a property only of the current implementation; grammars could be analyzed to detect these situations.

3.2. Syntax

A syntax extension can be used to define a completely new grammar, or to modify an existing one. There are two forms; a *syntax term*, and a (top-level) *syntax phrase*. We have seen examples of syntax phrases earlier. Here we start with syntax terms, which have the form:

`syntax` x_1`::=`g_1 `...` x_n`::=`g_n `in ... end`

The allowed forms for the g_i's were explained in the previous section. The resulting grammar (x_1) is then used to parse the *span* of the grammar, which is the input stream after '`in`'. If this parsing is successful, the keyword '`end`' is expected, and then the current grammar reverts to the one that was active before entering the syntax term.

The result of parsing a syntax term '*`s`*' is a '`term`' according to the basic F-sub syntax of terms (that is, where all the syntax extensions have been expanded). The expansion of a syntax term '*`s`*' into a '`term`' is directed by the *actions* that are defined in '*`s`*'; if no action is specified, '*`s`*' expands simply to '`top`'. For example, we define below a grammar with two possible parses, the keywords '`one`' and '`two`', and no actions. (We use outer brackets to avoid confusing a syntax term with a syntax phrase.)

```
- {syntax x::={"one" "two"} in one end};
⇨ top : Top
```

A quoted identifier like '`"one"`' is automatically made into a keyword in the relevant span. Keywords are inherited from outer spans to inner spans. (Hence the built-in F-sub keywords may conflict with syntax extensions.)

A top-level *syntax phrase* is a syntax term where the part '`in ... end`' is missing; its span is the remainder of the top-level session (but see Section 4).

A syntax phrase does not normally affect the immediate top-level syntax. That is, the non-terminal '`phrase`' given in Appendix C keeps being used for parsing at the '`- `' prompt.

But if the '`toplevel`' keyword is used, then the first non-terminal of the given grammar is adopted as the new top-level syntax, and the built-in F-sub syntax is completely bypassed:

```
- syntax toplevel x::={"one" "two"};
- one
⇨ top : Top
```

```
- two
⇨ top : Top
```

Note that we are now stuck with ‘`x`’ as the top-level syntax; see Section 4 for recovering from this situation.

Instead of defining a completely new grammar we can extend an existing one. In particular, we can extend the existing F-sub grammar described in Appendix C. Useful starting points for extension are ‘`termBase`’ and ‘`typeBase`’ (but use ‘`term`’ and ‘`type`’ on the right-hand side of ‘`::=`’). See Appendix C for other non-terminals that can be extended; these are marked *public*.

To extend a non-terminal ‘x’ bound to a grammar ‘g_1’ write:

$$x\ ::=\ \ldots\ g_2$$

In this case ‘x’ becomes equivalent to the choice ‘{g_1 g_2}’. In particular, ‘x`::=...{}`’ has no effect, while ‘x`::=...[]`’ makes ‘x’ optional. For example:

```
- {syntax termBase::= ... {"one" "two"}
   in {fun(x:Top)one}(two) end};
⇨ top : Top
```

The final topic of this section is how to add infix operators. This is achieved by extending grammars that begin with an iteration construct, as opposed to extending arbitrary grammars as shown above.

To extend the iteration part of a non-terminal ‘x’ bound to an iteration grammar ‘(g_1*g_2)’ write:

$$x\ ::=\ \ldots\ *\ g_3$$

Then ‘x’ becomes equivalent to the iteration ‘(g_1*{g_2 g_3})’.

In Appendix C we provide a non-terminal ‘`termOper`’ as a suitable place for attaching infix operators. This ‘`termOper`’ is an iteration based on ‘`termAppl`’. The latter is another iteration that parses applications, and is in turn based on ‘`termBase`’. Finally, ‘`termBase`’ terms are those simple terms that do not have pieces of syntax “hanging off to the right”. Given this structure, one can attach infix operators to ‘`termOper`’ that will have lower precedence than application.

The following iteration extension introduces ‘`+`’ as a left-associative infix operator over ‘`termOper`’s:

```
termOper ::= ... * [ "+" termAppl ]
```

achieving the equivalent of ‘`termOper ::= ( termAppl * [ "+" termAppl ] )`’.

The following iteration extension introduces ‘`-`’ as a right-associative infix operator over ‘`termOper`’s:

```
termOper ::= ... * [ "-" termOper ]
```

achieving the equivalent of ‘`termOper ::= ( termAppl * [ "-" termOper ] )`’.

Similarly, ‘`typeOper`’ and ‘`typeBase`’ can be used for new infix type operators (there is no ‘`typeAppl`’). The syntax of ‘`type`’ in Appendix C implies that these operators will have higher precedence than ‘`->`’.

3.3. Actions

Actions can be attached to grammars. They describe the terms that are to be generated during parsing of syntax extensions. By using ‘`g_n`’ in a grammar, one specifies that the result of parsing ‘`g`’ should be stored in the pattern variable ‘`_n`’. Pattern variables are then used in actions ‘`a`’; the grammars ‘`g=>a`’ and ‘`g:>a`’ specify, respectively, that the parsing of ‘`g`’ should generate the term or type described by ‘`a`’.

We now describe the rules of expansion. The *expansion* generated by a (successfully parsed) grammar is defined as follows:

Grammar	*Expansion*
`x`	the expansion generated by the grammar bound to `x`.
`ide`	`top`.
`"..."`	`top`.
`string`	`top`.
$[g_1 \ldots g_n]$	`top`.
$\{g_1 \ldots g_n\}$	the expansion generated by the successful g_i.
$(g_1 * g_2)$	the expansion generated by either g_1, if g_1 alone is successful, or by the last g_2 if $[g_1\ g_2 \ldots g_2]$ is successful.
`g=>a`	the expansion generated by the *term pattern* `a` (see below).
`g:>a`	the expansion generated by the *type pattern* `a` (see below).
`g_n`	`top`, but in addition the expansion generated by `g` is stored in `_n`.
$(g_1$ `*_n` $g_2)$	the expansion generated by $(g_1 * g_2)$, but at each iteration the latest expansion is also stored in `_n`.

A pattern variable ‘`_n`’ (with `n` non-negative) is *defined* when an occurrence of ‘`g_n`’ is parsed successfully. The *range of definition* of a pattern variable ‘`_n`’ is always confined within a clause ‘`x::=g`’. In addition, a pattern variable defined in a branch of a choice is confined to that branch, and one defined in the ‘g_2’ of ‘$(g_1 * g_2)$’ is confined to ‘g_2’. Errors are given on attempts to define a pattern variable twice, or to use one that is not currently defined.

An *action* ‘`a`’ may contain the pattern variables ‘`_n`’ that are defined where ‘`a`’ appears. Note that an action ‘`a`’ in ‘`g=>a`’ can access pattern variables defined outside ‘`g`’ in the surrounding grammar; this ability greatly increases the expressive power of actions. An action may also contain ordinary program variables bound in the surrounding scope.

An action appearing after ‘`=>`’ can be any *term pattern*. This is, recursively, either a ‘`term`’ (including any syntax extension of ‘`term`’) or one of the following patterns:

`_n`
`fun(_n:`*type-pattern*`)` *term-pattern*
`fun(_n<:`*type-pattern*`)` *term-pattern*
`fun(_n)` *term-pattern*

The *expansion* generated by a term pattern is the result of instantiating the term pattern with the expansions stored in the pattern variables ‘`_n`’ that occur in it.

Similarly to term patterns, an action appearing after ‘`:>`’ can be any *type pattern*, which is, recursively, either a ‘`type`’ or one of the following patterns:

`_`*n*
`All(_`*n*`<:`*type-pattern*`)` *type-pattern*
`All(_`*n*`)` *type-pattern*

We are careful to avoid variable capture when patterns are instantiated. Considering ‘`fun(`...`:`*A*`)` *b*’ binders (the others are handled similarly), we have the typical situations:

(1) `fun(x:`*A*`)x(_1)` (2) `fun(_1:`*A*`)x(_2)`

In ‘`fun(x:`*A*`)`’ situations, including example (1), the variable ‘`x`’ is consistently renamed so that it does not capture other variables named ‘`x`’ when the pattern is instantiated. In ‘`fun(_1:`*A*`)`’ situations, variable capture on instantiation is normally desired, but only for certain subexpressions. In example (2) we never want the variable that replaces ‘`_1`’ to capture ‘`x`’, but we always want the variable that replaces ‘`_1`’ to capture the similarly named variables in the term that replaces ‘`_2`’. The general situation is handled by two separate renaming environments during instantiation; one for *resident* bound variables (‘`x`’, in (1)) and one for *intruding* bound variables (the ones replacing ‘`_1`’ in (2)). Different subexpressions of the pattern are renamed according to the appropriate environment. Variables that are free in an action and bound in the top-level environment are allowed, but may produce error messages later when in risk of being captured.

3.4. How it is done

The implementation of syntax extensions is really quite simple, when properly organized. Grammars are stored in tables associating non-terminal names to grammar descriptions; this association can be changed dynamically to extend existing non-terminals. Grammar descriptions include client “action procedures” to be invoked during parsing to build the abstract syntax trees: no intermediate parse trees are built, resulting in very efficient parsing. Intermediate parsing results are kept on a stack, accessed by (the equivalent of) pattern variables.

A simple recursive-descent parser interprets these grammar tables blindly, dispatching on the various cases of grammar descriptions and calling the action routines when indicated. The action routines attached to the built-in syntax of grammars build grammars. The action routines attached to the syntax of actions, invoke an external “Act” interface to instantiate patterns. Nothing in this parser and syntax-extension machinery is specific to the implementation of F-sub; in fact, it could be and has been reused for other languages.

The built-in F-sub syntax is just a grammar table, so that it can be modified like any other grammar. The only parsing code specific to F-sub is provided in the implementation of the interface “Act”, used by the parser to instantiate the pattern variables within term and type patterns. This module is responsible for preventing variable captures, and hence must be aware of the scoping structures of the language at hand.

The sophisticated hiding and sharing of information needed to separate the parser from the rest of the system, is realized via the Modula-3 partially-opaque-types mechanism.

We now discuss in more detail how *actions* are instantiated so that variable capture is avoided. The basic technique is described in the simplified context of a λ-calculus with *λ-patterns*. The technique is then instatiated three times in F-sub, for ‘`All(`*X*`<:`*A*`)`’, ‘`fun(`*X*`<:`*A*`)`’, and ‘`fun(`*x*`:`*A*`)`’ binders.

A *pattern* p is described by the following data structure:

$$p = x \mid \lambda x.p \mid p\ p' \mid \underline{x} \mid \lambda\underline{x}.\ p$$

where the *pattern variables* $\underline{x}$ (corresponding to ‘_*n*’ in F-sub) are distinct from the ordinary variables x.

We use *renamings* ρ mapping (non-pattern-) variables to (non-pattern-) variables, and *instantiations* π, mapping pattern-variables to patterns. Here are the corresponding data structures and related operations:

$$\rho = \varepsilon \mid x{\leftarrow}y,\rho' \qquad x \notin \mathrm{dom}(\rho')$$
$$\pi = \varepsilon \mid \underline{x}{\leftarrow}p,\pi' \qquad \underline{x} \notin \mathrm{dom}(\pi')$$

$\mathrm{dom}(\rho)$: *domain*

$$\mathrm{dom}(\varepsilon) = \emptyset$$
$$\mathrm{dom}(x{\leftarrow}y,\rho) = \{x\} \cup \mathrm{dom}(\rho)$$

$\mathrm{rng}(\rho)$: *range*

$$\mathrm{rng}(\varepsilon) = \emptyset$$
$$\mathrm{rng}(x{\leftarrow}y,\rho) = \{y\} \cup \mathrm{rng}(\rho)$$

$\mathrm{dom}(\pi)$:

$$\mathrm{dom}(\varepsilon) = \emptyset$$
$$\mathrm{dom}(\underline{x}{\leftarrow}p,\pi) = \{\underline{x}\} \cup \mathrm{dom}(\pi)$$

$\mathrm{rng}(\pi)$:

$$\mathrm{rng}(\varepsilon) = \emptyset$$
$$\mathrm{rng}(\underline{x}{\leftarrow}p,\pi) = \{p\} \cup \mathrm{rng}(\pi)$$

$\rho(x)$: *lookup*

$$\varepsilon(x) = x$$
$$(z{\leftarrow}y,\rho)(x) = \rho(x) \qquad (z \neq x)$$
$$(x{\leftarrow}y,\rho)(x) = y$$

$\pi(\underline{x})$:

$$\varepsilon(\underline{x}) = \underline{x}$$
$$(\underline{z}{\leftarrow}p,\pi)(\underline{x}) = \pi(\underline{x}) \qquad (\underline{z} \neq \underline{x})$$
$$(\underline{x}{\leftarrow}p,\pi)(\underline{x}) = p$$

$\rho \backslash x$: *restriction*

$$\varepsilon \backslash x = \varepsilon$$
$$(z{\leftarrow}y,\rho) \backslash x = z{\leftarrow}y,\rho \backslash x \qquad (z \neq x)$$
$$(x{\leftarrow}y,\rho) \backslash x = \rho$$

$\pi \backslash \underline{x}$:

$$\varepsilon \backslash \underline{x} = \varepsilon$$
$$(\underline{z}{\leftarrow}p,\pi) \backslash \underline{x} = \underline{z}{\leftarrow}p,\pi \backslash \underline{x} \qquad (\underline{z} \neq \underline{x})$$
$$(\underline{x}{\leftarrow}p,\pi) \backslash \underline{x} = \pi$$

With these operations, we can define the notions of *free variables* (FV), *pattern variables* (PV), and *binding pattern variables* (BPV) of a pattern.

$\mathrm{FV}(p)$:

$$\mathrm{FV}(x) = \{x\}$$
$$\mathrm{FV}(\lambda x.p) = \mathrm{FV}(p) - \{x\}$$
$$\mathrm{FV}(p\ p') = \mathrm{FV}(p) \cup \mathrm{FV}(p')$$
$$\mathrm{FV}(\underline{x}) = \emptyset$$
$$\mathrm{FV}(\lambda \underline{x}.\ p) = \mathrm{FV}(p)$$

$\mathrm{PV}(p)$:

$$\mathrm{PV}(x) = \emptyset$$
$$\mathrm{PV}(\lambda x.p) = \mathrm{PV}(p)$$
$$\mathrm{PV}(p\ p') = \mathrm{PV}(p) \cup \mathrm{PV}(p')$$
$$\mathrm{PV}(\underline{x}) = \{\underline{x}\}$$
$$\mathrm{PV}(\lambda \underline{x}.\ p) = \{\underline{x}\} \cup \mathrm{PV}(p)$$

$\mathrm{BPV}(p)$:

$$\mathrm{BPV}(x) = \emptyset$$
$$\mathrm{BPV}(\lambda x.p) = \mathrm{BPV}(p)$$
$$\mathrm{BPV}(p\ p') = \mathrm{BPV}(p) \cup \mathrm{BPV}(p')$$
$$\mathrm{BPV}(\underline{x}) = \emptyset$$
$$\mathrm{BPV}(\lambda \underline{x}.\ p) = \{\underline{x}\} \cup \mathrm{BPV}(p)$$

Free variables and pattern variables are then extended to renamings and instantiations.

$\mathrm{FV}(\rho)$:

$$\mathrm{FV}(\rho) = \mathrm{rng}(\rho)$$

$\mathrm{PV}(\rho)$:

$$\mathrm{PV}(\rho) = \emptyset$$

$\mathrm{FV}(\pi)$:

$$\mathrm{FV}(\pi) = \bigcup\{\mathrm{FV}(p) \mid p \in \mathrm{rng}(\pi)\}$$

$\mathrm{PV}(\pi)$:

$$\mathrm{PV}(\pi) = \bigcup\{\mathrm{PV}(p) \mid p \in \mathrm{rng}(\pi)\}$$

Finally, we define the effect of applying renamings and instantiations to patterns.

$p[\rho]$:

$$x[\rho] = \rho(x)$$
$$(\lambda x.p)[\rho] = \lambda x'.p[x{\leftarrow}x',\rho \backslash x] \qquad x' \notin \mathrm{FV}(p),\ x' \notin \mathrm{dom}(\rho) \cup \mathrm{rng}(\rho)$$
$$(p\ p')[\rho] = p[\rho]\ p'[\rho]$$
$$\underline{x}[\rho] = \underline{x}$$
$$(\lambda \underline{x}.p)[\rho] = \lambda \underline{x}.p[\rho]$$

$p[\pi]$: assuming $\underline{x} \in BPV(p) \Rightarrow \pi(\underline{x})$ is a variable y

$$p[\pi] = p[\varepsilon; \varepsilon; \pi]$$

$$x[\rho; \rho'; \pi] = x[\rho]$$

$$(\lambda x.p)[\rho; \rho'; \pi] = \lambda x'.p[x \leftarrow x', \rho \backslash x; \rho'; \pi] \qquad x' \notin FV(p),\ x' \notin [\rho; \rho'; \pi]$$

$$(p\ p')[\rho; \rho'; \pi] = p[\rho; \rho'; \pi]\ p'[\rho; \rho'; \pi]$$

$$\underline{x}[\rho; \rho'; \pi] = \pi(\underline{x})[\rho']$$

$$(\lambda \underline{x}.p)[\rho; \rho'; \pi] = \lambda y'.\ p[\rho; \pi(\underline{x}) \leftarrow y', \rho' \backslash \pi(\underline{x}); \pi] \qquad y' \notin FV(p),\ y' \notin [\rho; \rho'; \pi]$$

where $x \notin [\rho; \rho'; \pi] \Leftrightarrow x \notin FV(\pi) \cup dom(\rho') \cup rng(\rho') \cup dom(\rho) \cup rng(\rho)$.

In $p[\rho; \rho'; \pi]$, we use ρ to rename the bound variables found in p, and we use ρ' to rename the variables found in the range of π that are placed in binding positions.

We now discuss these definitions and the reasons for their side-conditions.

Although eventually we must obtain a *ground* pattern (free of pattern variables) for evaluation, we cannot require that every pattern instantiation immediately produces a ground pattern. This is because, in order to define new syntax extensions in terms of old ones, extended syntax may appear in actions. For example, consider the syntax extensions and actions ‘`[x "+" y] => plus(x)(y)`’ and ‘`[x "avg" y] => div(y+x)(two)`’. When parsing the latter action we have a non-ground instantiation of ‘`div(y+x)(two)`’ to ‘`div(plus(y)(x))(two)`’. Only later, when ‘`one avg three`’ is met, we obtain a ground pattern ‘`div(plus(tree)(one))(two)`’.

However, we cannot allow arbitrary non-ground instantiations. Of course, we cannot replace a binding pattern variable with, for example, a λ-abstraction. But in addition, it seems we cannot replace a binding pattern variable with another pattern variable; otherwise we could write: $(\lambda \underline{x}.\lambda \underline{y}.\underline{x}\underline{y})[\underline{x} \leftarrow \underline{z}, \underline{y} \leftarrow \underline{z}, \varepsilon] = (\lambda \underline{z}.\lambda \underline{z}.\underline{z}\underline{z})$, which causes a pattern-variable capture. This justifies the restriction ($\underline{x} \in BPV(p) \Rightarrow \pi(\underline{x})$ is a variable y) in the definition of $p[\pi]$. Note that binding pattern variables cannot be α-converted because they are “visible from the outside”.

The informal idea that “there are no variable captures” should be formalized by showing that the renaming or instantiation of α-equivalent patterns produces α-equivalent patterns, and by deriving expected properties of substitutions. We leave this for future work.

3.5. Related work on syntax extension

Griffin [Griffin 1988] has enumerated desirable properties of notational definitions and has studied their formalization. Our distinction between normal λ’s and pattern-λ’s seems to remain implicit in his work. Unlike Griffin, who translates to combinator forms that then reduce to the desired programs, we synthesize those programs directly. (Griffin would handle our ‘`let x=a in b end`’ example by translating to ‘`LET(λx.b)(a)`’ for an appropriate combinator ‘`LET`’.) Moreover, while Griffin discusses abstract translations, we provide a specific grammar definition technique and an efficient parsing algorithm. Parsing is efficient because it is LL(1) and because it avoids the creation of intermediate parse trees, producing directly abstract syntax trees that do not require normalization.

Bove and Arbilla [Bove, Arbilla 1992] discuss how to use explicit substitutions to implement syntax extensions. This is an elegant idea that we could perhaps have adopted, but we managed to work with ordinary substitutions over de Bruijn indices. As in the previous case, their work does not describe a parsing algorithm, but is theoretically well developed.

Some language implementations, like CAML and SML, integrate a YACC or similar parser generator that allows them to introduce new syntax [Mauny, Rauglaudre 1992]. If the new syntax is to be mixed with the old one, the new syntax must be quoted in some way. Instead, we can freely intermix new and old syntax without special quotations.

Hygienic macros [Kohlbecker, *et al.* 1986] share many of the same goals as our syntax extensions; however, these macros account only for macro calls and not for liberally introducing new syntax. Hygienic macros employ a multiple-pass time-stamping algorithm to prevent variable capture; this algorithm is, at least operationally, different from our single-pass multiple-environment algorithm. We do not handle quotation and antiquotation in the style of Lisp.

Finally, our syntax extension mechanism guarantees termination of parsing, even when our "macros" are recursively defined. This property does not hold for many macro mechanisms that are computationally powerful.

4. Mock-modules and save-points

A crude modularization mechanism is provided as an aid to the interactive loading and reloading of definitions. Separate compilation is not a goal.

To facilitate loading and reloading the file, say, '`One.fsub`' containing F-sub definitions, one should start that file with the following phrase (the module name must be the same as the file name):

```
module One;
```

If this file relies on definitions contained in files '`Two.fsub`' and '`Three.fsub`' (which should in turn start with the lines '`module Two;`' and '`module Three;`' respectively) then '`One.fsub`' should start with the phrase :

```
module One import Two Three;
```

Then the variables defined inside `Two` and `Three` become available within `One`.

A *reload phrase* can be issued at the top-level to load or to force reloading a module. ('`load`', which was briefly discussed in section 2, will not reload a module that is already loaded.):

```
- reload One;
```

The meaning of '`reload One;`' is simply to read the Unix file '`./One.fsub`'. A quoted string can also be placed after '`reload`', in which case the indicated file name is used without modification.

The intent of *re*loading a (file containing a) module, is to backtrack to the point in time when that module was first loaded. All the intervening top-level definitions (including syntax extensions) are retracted. When reloading a module, only the imported modules that are not already present are reloaded; in particular, a module imported through two different import paths is loaded once.

The precise behavior of this module mechanism is now described in terms of some lower-level primitives that handle *save-points*. In contrast to module phrases, which are mostly useful when used within files, save-points may be useful also when interacting at the top-level. For example, they are available even when the top-level syntax has been clobbered by the syntax extension mechanism.

A *save-point* is a record of the complete state of the system at a given point in time.

```
- save that;
```

This phrase creates a save-point called, in this case, ‘`that`’, recording the state of the system at the moment it is issued. Named save-points are stacked.

Later, one can issue the phrase:

```
- restore that;
```

which resets the system back to ‘`that`’ save-point, possibly obliterating top-level definitions as well as intervening save-points with different names. The save-point ‘`that`’ is, however, maintained.

A special save-point exists in the beginning; the phrase:

```
- restore;
```

restores the system to its initial condition just after start-up.

```
- establish that;
```

This phrase is equivalent to ‘`save that;`’, if a save-point called ‘`that`’ does not exist, and to ‘`restore that;`’, if a save-point called ‘`that`’ does exist.

```
- load that;
```

This is equivalent to ‘`reload that;`’ (that is, just reading the file) if a save-point called ‘`that`’ does not exist, but is a no-op if a save-point called ‘`that`’ already exists.

We can now describe the precise meaning of ‘`module`’ phrases. A module of the form:

```
module One import Two Three; ...
```

is simply treated as the sequence:

```
load Two; load Three; establish One; ...
```

where the ‘`load`’ phrases may end up establishing the corresponding modules because of module phrases in the loaded files.

5. Top-level phrases

The top-level phrases fall into several classes. We have described mock-modules, save-points and loading in Section 4. We now expand on the definition and evaluation phrases sketched in Section 2. Moreover, we discuss judgment phrases and command phrases.

All the phrases that involve types or terms are elaborated as follows. The *parsing phase* expands the syntax extensions. Then, a *scoping phase* expands type definitions, converts identifiers to de Bruijn indices, and detects unbound identifiers. Next, a *checking phase* verifies the typing correctness of types and terms. Then, an *evaluation phase* normalizes terms. Finally, a *printing phase* prints the results; identifiers with the same name but different de Bruijn indices are decorated in different ways. If an error occurs in one of these phases, the file name (if any) and the line position of the error is reported.

Each phrase is elaborated in the context of the previous top-level phrases.

- A *type definition phrase* has the form:

$$\texttt{- Let } X_1\texttt{<:}A_1\texttt{ = }B_1 \ldots X_n\texttt{<:}A_n\texttt{ = }B_n\texttt{;}$$

where the bounds '$\texttt{<:}A_i$' can be omitted, with 'A_i' defaulting to '`Top`'. Each 'A_{i+1}' and 'B_{i+1}' is in the scope of 'X_1' ... 'X_i' and of all the previous top-level definitions. Type definitions are fully expanded before typechecking.

- A *term definition phrase* has the form:

    $\texttt{- let } x_1\texttt{:}A_1 \texttt{ = } b_1 \dots x_n\texttt{:}A_n \texttt{ = } b_n\texttt{;}$

where the domains '$\texttt{:}A_i$' can be omitted, with 'A_i' being inferred from 'b_i'. Each 'A_{i+1}' and 'b_{i+1}' is in the scope of 'x_1' ... 'x_i' and in the scope of all the previous top-level definitions.

- A *type phrase* has the form:

    `- :A;`

which results in checking the type 'A' with respect to the current top-level definitions.

- A *term phrase* has the form:

    `- a;`

which results in checking and evaluating the term 'a' with respect to the current top-level definitions.

- An *environment* E (often called also a *context* or an *assignment*) is a possibly empty sequence of either type variables with a bound ('$X\texttt{<:}A$') or term variables with a domain ('$x\texttt{:}A$'). Each variable is in the scope of the environment to its left and in the scope of the top-level definitions.

- A *judgment* is one of the four formal statements axiomatized in Appendix D, each involving an environment. Each of the four statements has a corresponding phrase, as follows.

An *environment judgment phrase* has the form:

`- judge env E;`

where the environment 'E' is in the scope of the previous top-level definitions (and similarly for the following judgments).

A *type judgment phrase* has the form:

`- judge type E |- A;`

A *subtype judgment phrase* has the form:

`- judge subtype E |- A<:B;`

Finally, a *term judgment phrase* has the form:

`- judge term E |- a:A;`

If the correctness of one of these judgments is established, a simple '`ok`' is printed. It is informative to turn on tracing (as described below) when elaborating judgments.

- A *command phrase* is used to switch on and off various options. It has the form:

    `- do command argument;`

One can get a list of all the available commands by writing:

`- do;`

and one can find out about an individual command by writing:

`- do command;`

The command '`do Version;`' prints the current version of the system.

After issuing the command '`do ShowParsing On;`' the result of parsing each phrase is printed. This is useful for debugging syntax extensions.

After issuing the command '`do ShowVarIndex On;`' the de Bruijn indices of variables are printed along with the variables.

The command '`do QuantifierSubtyping` *X*`;`' switches between the undecidable *F<:* rule for quantifier subtyping (*X* = `LeastBound`), the decidable Fun rule [Cardelli, Wegner 1985] (*X* = `EqualBounds`) and a decidable rule proposed by Giuseppe Castagna (*X* = `TopBound`).

After issuing the command '`do TraceType On;`', each call to the *type* routine of Appendix E is traced. Similarly, '`do TraceSubtype On;`' and '`do TraceTerm On;`' correspond to the *sub* and *term* routines.

Some other commands are used for system debugging and are not documented here.

6. Type inference by "argument synthesis"

In pure F-sub one has to write down an often overwhelming amount of type information. This is already evident when encoding something as simple as pairing constructs. For example, through syntax extensions we can define a cartesian product operator '`A*B`' as '`All(C){A->B->C}->C`' (see Appendix A), along with the operations:

```
pair: All(A)All(B)A->B->A*B
fst: All(A)All(B)A*B->A
snd: All(A)All(B)A*B->B
```

To create and manipulate simple pairs we have to write, for example;

```
- let a = pair(Bool)(Top)(true)(top);    (* the pair (true,top) *)
- fst(Bool)(Top)(a);             (* the first component of pair a *)
```

A triple is already quite a challenge:

```
- pair(Bool)(Top*Bool)(true)(
    pair(Top)(Bool)(top)(false)); (* the triple (true,top,false)*)
```

What is worse, we cannot even define a syntax extension of the form, for example, '*a*`,`*b*' for pairs, because the type arguments must be provided somehow.

Fortunately, a form of type inference is available. To enable it, we append question marks '`?`' to the type parameters that we would like to omit (loosely following [Pollack 1990]). For example, the polymorphic identity could be written:

```
- Let Id = All(X?)X->X;
- let id: Id = fun(X?)fun(x:X)x;
```

Then, the type arguments corresponding to question-mark parameters must be omitted:

```
- id(top);  (* instead of  id(:Top)(top) *)
⇨ top : Top
```

In this situation, we say that the type parameter '`X`' of '`Id`' is *stripped*, to compensate for the missing type argument, and that the argument is later *synthesized*.

A type quantifier is stripped by introducing a fresh *unification variable* that may be instatiated later, or never; a unification algorithm is responsible for the synthesis of the argument. Type parameters are stripped if and only if they appear at the beginning of the type of a term identifier (that is, not an arbitrary term): we found this restriction useful both for the inference algorithm and in understanding how inference behaves in actual programs. Here is a situation where stripping occurs, and a unification variable is exposed in the printed result:

```
- id;
⇨ {fun(x:X?)x} : {X?->X?}
```

If needed, we can prevent stripping by placing an exclamation mark after a term identifier:

```
- id!;
⇨ <id> : <Id>
```

This option is useful, for example, if we want to pass the (unstripped) polymorphic identity as an argument to another term.

Going back to pairs, we can now rewrite our primitives so that they admit type inference:

```
pair: All(A?)All(B?)A->B->A*B
fst: All(A?)All(B?)A*B->A
snd: All(A?)All(B?)A*B->B
```

This allows us to write triples a bit more compactly, by omitting the type arguments:

```
- pair(true)(pair(top)(false)); (* the triple (true,top,false) *)
```

But, what is more important, we can now put syntax extensions to work and define a simple '`a,b`' notation:

```
- syntax
    termOper ::= ... *_1
      [ "," termOper_2 ] => pair(_1)(_2);

- true,top,false;                  (* the triple (true,top,false) *)
```

We are finally able to write pairs in a convenient notation, by the interplay of type inference and syntax extensions.

We conclude this section with some general remarks about this form of type inference; details of the algorithm are in Appendix F.

The types '`All(X?<:A)B`' and '`All(X<:A)B`' are incomparable. A type '`All(X?<:Top)B`' is stripped to '`B`' where '`X`' is treated as a fresh unification variable. Instead, a type '`All(X?<:A)B`' with a non-'`Top`' bound is stripped simply to '`B{X←A}`'.

When an occurrence of '`X`' bound by an '`X?`' appears nested within other quantifiers, it must not be instantiated in a way that will cause variable captures. To this end, we used *first-order unification under a mixed prefix* [Miller (to appear)]. For a practical example of where this matters, see the Existentials section in Appendix A. As an ad-hoc example, consider the following term where a type parameter is omitted in the application of `f`:

```
- fun(f:All(Y?){All(W)W->Y}->Y) f(fun(Z)fun(z:Z)z);
⇨ Type error. Type inference rank check: instantiation type for
   Y? contains a (different) variable Z that is bound deeper
   than the Y? binder:
      Z(=W)  (last input line, char 46)
   Error detected (last input line, char 51)
```

If ordinary unification is used instead of mixed-prefix unification, we match `All(W)W->Y?` against `All(Z)Z->Z`, causing the unification of `Y?` with `Z`. Hence the whole term above acquires the type:

```
{All(Y?){All(W)W->Y}->Y} -> Z
```

where the final `Z` (which is unified with `Y`) has escaped its scope and remains unbound. We can then provide the following argument for the term above, obtaining a term that has the escaped Z as its type:

```
fun(Y?)fun(g:All(W)W->Y)g(Top)(top)
```

After typechecking an entire top-level phrase, some of the unification variables used for type inference may remain undetermined. We choose to tolerate this situation in term-phrases (‘*a*`;`’), but we report an error in term-definition-phrases (‘`let`...`;`’).

We believe that our type inference algorithm is essentially the same as the one used in LEGO [Pollack 1990] and a first-order version of the one used in ELF [Pfenning 1989] (although we have no detailed knowledge of those implementations). We believe the algorithm is sound, but is not complete, particularly because we are using unification in a subtyping context. As a heuristic, this inference algorithm works exceedingly well.

7. Recursion

In this section we describe an extension of F-sub with recursive types and recursive values. The integration of recursion with subtyping in a first-order system is studied in [Amadio, Cardelli 1991]. The ideas described there should work in a second-order system such as *F<:* . However, here we take a simpler approach to recursive types, to minimize their interference with second-order types and type inference techniques.

The main idea is that the isomorphism between a recursive type ‘`Rec(`*X*`)`*B*’ and its unfolding ‘*B*{*X*←`Rec(`*X*`)`*B*}’ is made explicit in the syntax of terms. In first approximation, we have:

```
unfold : Rec(X)B -> B{X←Rec(X)B}
fold   : B{X←Rec(X)B} -> Rec(X)B
```

More precisely, we extend the syntax of F-sub as follows:

A`,`*B* `::=` `...`	types as before, plus:
`Rec(`*X*`)`*B*	recursive types
a`,`*b* `::=` `...`	terms as before, plus:
`fold(:`*A*`)(`*b*`)`	fold *b* into an element of the recursive type *A*
`unfold(`*b*`)`	unfold an element *b* of a recursive type
`rec(`*x*`:`*A*`)`*b*	recursive terms

Since the isomorphism is explicit, we do *not* have ‘$\mathtt{Rec}\,(X)\,B$’ = ‘$B\{X{\leftarrow}\mathtt{Rec}\,(X)\,B\}$’. Instead, two recursive types are equal only if their respective ‘`Rec`’ binders are found in corresponding positions. Given this restriction, the recursive subtyping algorithm becomes much simpler (while remaining non-trivial). The central type rule for recursive subtyping is unchanged, but the auxiliary judgment and rules having to do with type equality [Amadio, Cardelli 1991] are dropped. The type rules and algorithms are described in Appendix G.

As a simple use of recursive types, let us define the type of untyped lambda-terms, and some standard combinators.

```
Let V = Rec(V) V->V;

let lam: {V->V}->V = fun(f:V->V) fold(:V)(f)
    app: V->{V->V} = fun(f:V) fun(a:V) unfold(f)(a);

let i: V = lam(fun(x:V)x)
    k: V = lam(fun(x:V) lam(fun(y:V) x))
    s: V = lam(fun(x:V) lam(fun(y:V) lam(fun(z:V)
             app(app(x)(z))(app(y)(z)))));
let y: V = rec(y:V) lam(fun(f:V) app(f)(app(y)(f)));
```

With a bit of syntax extension one can eliminate the ‘`lam`’ and ‘`app`’ clutter (see Appendix A).

Appendices

Appendix A. Examples

Identity

This is the file '`Id.fsub`'. It defines the polymorphic identity in such a way that its type parameter can be omitted.

```
module Id;

Let Id = All(X?) X->X;

let id: Id = fun(X?) fun(x:X) x;
```

Unit

This is the file '`Unit.fsub`'. '`Unit`' is the encoding of a data type with a single element '`unit`'. It is essentially the same as the polymorphic identity, but because of the intended use of '`unit`', type inference is not desirable.

```
module Unit;
(* Defines:
    Unit = All(X)X->X
    unit: Unit
*)

Let Unit = All(X) X->X;

let unit: Unit = fun(X) fun(x:X) x;
```

Booleans

This is the file '`Bool.fsub`'. This is the encoding of a data type with two elements, '`true`' and '`false`'. Also provided are two subtypes of '`Bool`' containing one element each. Standard boolean operators are defined. The syntax of terms is extended with two keywords '`true`' and '`false`', with conditionals, and with two infix operators.

```
module Bool;
(* Defines:
    Bool = All(X)X->X->X
    True, False <: Bool
    true, false: Bool
    tt: True
    ff: False
    not: Bool->Bool
    and, or, _/\_, _\/_: Bool->Bool->Bool
    if _ then _ else _ end
*)

Let Bool = All(X) X->X->X
    True = All(X) X->Top->X
    False = All(X) Top->X->X;

let true: Bool = fun(X) fun(x:X) fun(y:X) x
    false: Bool = fun(X) fun(x:X) fun(y:X) y;

let tt: True = fun(X) fun(x:X) fun(y:Top) x
    ff: False = fun(X) fun(x:Top) fun(y:X) y;

let cond = fun(X?) fun(b:Bool) b(:X);

(* Bool, true, and false are turned into keywords *)
```

```
syntax
  typeBase ::= ...
    "Bool" :> Bool

  termBase ::= ...
    { "true" => true
      "false" => false
      ["if" term_1 "then" term_2 "else" term_3 "end" ]
        => cond(_1)(_2)(_3) }
;

let not: Bool->Bool =
      fun(x:Bool) if x then false else true end
    and: Bool->Bool->Bool =
      fun(x:Bool) fun(y:Bool)
        if x then y else false end
    or: Bool->Bool->Bool =
      fun(x:Bool) fun(y:Bool)
        if x then true else y end;

syntax
  termOper ::= ... *_1
    { [ "/\\" termAppl_2 ] => and(_1)(_2)
      [ "\\/" termAppl_2 ] => or(_1)(_2) }
;
```

Products

This is the file ‘`Product.fsub`’. It defines a cartesian product operator, extending the syntax of types, and a pairing operator, extending the syntax of terms. Syntax extensions and type inference interact in this situation, so that pairs can be constructed simply by infixing a ‘`,`’.

```
module Product;
(* Defines:
    A*B = All(C){A->B->C}->C
    _,_: All(A?) All(B?) A->B->A*B
    pair: All(A?) All(B?) A->B->A*B
    fst: All(A?) All(B?) A*B->A
    snd: All(A?) All(B?) A*B->B
*)

syntax
  typeOper ::= ... *_1
    ["*" typeOper_2]
      :> All(C) {_1->_2->C}->C
;

let pair: All(A?) All(B?) A->B->A*B =
  fun(A?) fun(B?) fun(a:A) fun(b:B)
    fun(C) fun(p:A->B->C) p(a)(b);

let fst: All(A?) All(B?) A*B->A =
  fun(A?) fun(B?) fun(p:A*B) p(:A)(fun(a:A)fun(b:B)a);
let snd: All(A?) All(B?) A*B->A =
  fun(A?) fun(B?) fun(p:A*B) p(:B)(fun(a:A)fun(b:B)b);

syntax
  termOper ::= ... *_1
    ["," termOper_2] => pair(_1)(_2)
;
```

Sums

This is the file ‘`Sum.fsub`’. It defines a disjoint union operator, extending the syntax of types, and a ‘`case`’ construct extending the syntax of terms. Note that ‘`case`’ introduces local bindings.

```
module Sum;
(* Defines
    A+B = All(C){A->C}->{B->C}->C
    inl: All(A?) All(B?) A->A+B
    inr: All(A?) All(B?) B->A+B
    sum: All(A?) All(B?) All(C?) A+B->{A->C}->{B->C}->C
    case term
      inl(ide:type) term
      inr(ide:type) term
    end,
*)

syntax
  typeOper ::= ... *_1
    [ "+" typeOper_2 ]
      :> All(C) {_1->C}->{_2->C}->C
;

let inl: All(A?) All(B?) A->A+B =
  fun(A?) fun(B?) fun(a:A)
    fun(C) fun(f:A->C) fun(g:B->C) f(a);
let inr: All(A?) All(B?) B->A+B =
  fun(A?) fun(B?) fun(b:B)
    fun(C) fun(f:A->C) fun(g:B->C) g(b);

let sum: All(A?) All(B?) All(C?) A+B->{A->C}->{B->C}->C =
  fun(A?) fun(B?) fun(C?)
    fun(s:A+B) fun(f:A->C) fun(g:B->C)
      s(:C)(f)(g);

syntax
  termBase ::= ...
      ["case" term_1
       "lft" "(" termIde_2 ":" type_3 ")" term_4
       "rht" "(" termIde_5 ":" type_6 ")" term_7
       "end"]
       => sum(_1)(fun(_2:_3)_4)(fun(_5:_6)_7)
;
```

Tuples

This is the file ‘`Tuple.fsub`’. It defines type tuples as iterated cartesian products ending with ‘`Top`’, so that a longer tuple type is a subtype of a shorter tuple type. Note that the previously defined syntax for cartesian products is used here to provide a further syntax extension. Tuple values are iterated pairings ending with ‘`top`’.

```
module Tuple
import Product;
(* Defines:
    Tuple(type ... type)
    tuple(term ... term)
*)

syntax
  typeBase ::= ...
    [ "Tuple" "(" typeTuple_1 ")" ] :> _1
  typeTuple ::=
    { [ type_1 typeTuple_2 ] :> _1 * _2
      [] :> Top }
;

syntax
  termBase ::= ...
    [ "tuple" "(" termTuple_1 ")" ] => _1
  termTuple ::=
    { [ term_1 termTuple_2 ] => _1 , _2
      [] => top }
;
```

Inductive Lists

This is the file '`IndList.fsub`'. It defines '`List(A)`' data types encoded by inductive definitions (that is, without using recursion over types). Syntax extensions are used here to simulate a third-order operator ('`List`') within a second-order language: '`List(A)`' is a second-order type only for a fixed '`A`'. Note that the action for '`List`' uses a local variable '`L`' that must be kept distinct from any variable that may appear in a parameter to '`List`'; this is taken care of by the action instantiation algorithm. The syntax of terms is extended with a case construct and a convenient way of building lists of many elements; again, syntax extensions and type inference interact in interesting ways.

```
module IndList
import Bool;
(* Defines:
    List(A) = All(L) L->{A->L->L}->L,
    nil: All(A?) List(A),
    cons: All(A?) A->List(A)->List(A),
    null: All(A?) List(A)->Bool,
    hd: All(A?) List(A)->A->A,
    tl: All(A?) List(A)->List(A)
    caseList term
      nil() term
      cons(ide:type ide:type) term
    end,
    list(term ... term)
*)

syntax
  typeBase ::= ...
    [ "List" "(" type_1 ")" ]
      :> All(L) L->{_1->L->L}->L
;

let nil: All(A?)List(A) =
  fun(A?)fun(L)fun(n:L)fun(c:A->L->L)n;

let cons: All(A?)A->List(A)->List(A) =
  fun(A?)fun(hd:A)fun(tl:List(A))
    fun(L)fun(n:L)fun(c:A->L->L)
      c(hd)(tl(:L)(n)(c));

let iterList: All(A?)All(B?) List(A)->B->{A->B->B}->B =
  fun(A?)fun(B?)fun(l:List(A))
    fun(n:B)fun(c:A->B->B)
      l(:B)(n)(c);

syntax
  termBase ::= ...
    { "nil" => nil
      "cons" => cons
      ["caseList" term_1
       "nil" "(" ")" term_2
       "cons" "(" termIde_3 ":" type_4 termIde_5 ":" type_6 ")" term_7
        "end"]
       => iterList(_1)(_2)(fun(_3:_4)fun(_5:_6)_7) }
;

let null: All(A?)List(A)->Bool =
  fun(A?)fun(l:List(A))
    caseList l
      nil() true
      cons(hd:A tl:Bool) false
    end;

let hd: All(A?)List(A)->A->A =
  fun(A?)fun(l:List(A))fun(a:A)
    caseList l
      nil() a
      cons(hd:A tl:A) hd
    end;
```

```
let tl: All(A?)List(A)->List(A) =
  fun(A?)fun(l:List(A))
    caseList l
      nil() nil
      cons(hd:A tl:List(A)) tl
    end;

syntax
  termBase ::= ...
    [ "list" "(" termList_1 ")" ] => _1
  termList ::=
    { [ term_1 termList_2 ] => cons(_1)(_2)
      [] => nil }
;
```

Recursive Lists

This is the file ‘`RecList.fsub`’. It provides the same constructions as ‘`IndList.fsub`’, except that lists are encoded via recursive types. Note how the operators provided here encapsulate the folding and unfolding of recursion, so that they need not be used directly.

```
module RecList
import Bool;
(* Defines:
    List(A) = Rec(L) All(C) C->{A->L->C}->C,
    nil: All(A?) List(A),
    cons: All(A?) A->List(A)->List(A),
    null: All(A?) List(A)->Bool,
    hd: All(A?) List(A)->A->A,
    tl: All(A?)List(A)->List(A),

    caseList term
      nil() term
      cons(ide:type ide:type) term
    end,
    list(term ... term)
*)

syntax
  typeBase ::= ...
    [ "List" "(" type_1 ")" ]
      :> Rec(L) All(C) C->{_1->L->C}->C
;

let nil: All(A?)List(A) =
  fun(A?)
    fold(:List(A))(fun(C)fun(n:C)fun(c:A->List(A)->C)n);

let cons: All(A?)A->List(A)->List(A) =
  fun(A?)fun(hd:A)fun(tl:List(A))
    fold(:List(A))(fun(C)fun(n:C)fun(c:A->List(A)->C)c(hd)(tl));

let recList: All(A?) All(B?) B->{A->List(A)->B}->List(A)->B =
  fun(A?) fun(B?) fun(n:B) fun(c:A->List(A)->B)
    fun(l:List(A)) unfold(l)(:B)(n)(c);

syntax
  termBase ::= ...
    { "nil" => nil
      "cons" => cons
      ["caseList" term_1
       "nil" "(" ")" term_2
       "cons" "(" termIde_3":"type_4 termIde_5":"type_6 ")" term_7
       "end"]
         => recList(_2)(fun(_3:_4)fun(_5:_6)_7)(_1)
    };
```

```
let null: All(A?)List(A)->Bool =
  fun(A?)fun(l:List(A))
    caseList l
      nil() true
      cons(hd:A tl:List(A)) false
    end;

let hd: All(A?)List(A)->A->A =
  fun(A?)fun(l:List(A))fun(a:A)
    caseList l
      nil() a
      cons(hd:A tl:List(A)) hd
    end;

let tl: All(A?)List(A)->List(A) =
  fun(A?)fun(l:List(A))
    caseList l
      nil() nil
      cons(hd:A tl:List(A)) tl
    end;

syntax
  termBase ::= ...
    [ "list" "(" termList_1 ")" ] => _1
  termList ::=
    { [ term_1 termList_2 ] => cons(_1)(_2)
      [] => nil }
;
```

Existentials

This is the file ‘`Some.fsub`’. Bounded and unbounded existential quantifiers are encoded in terms of universal quantifiers. Syntax is provided which is analogous to the built-in syntax for universal quantification.

```
module Some;
(* Defines
    Some(ide)type, Some(ide<:type)type
    pack ide<:type=type as type with term end
    open term as ide<:type ide:type in term end
*)

(* easy version:
syntax
  typeBase ::= ...
    [ "Some" "(" typeIde_1 "<:" type_2 ")" type_3 ]
      :> All(V?) {All(_1<:_2) _3 -> V} -> V
; *)

(* some interesting pattern-variable manipulation: *)
syntax
  typeBase ::= ...
    [ "Some" "(" typeIde_1
      { ["<:" type_2 ")" type_3 ]
      :> All(V?) {All(_1<:_2) _3 -> V} -> V
        [")" type_3 ]
      :> All(V?) {All(_1) _3 -> V} -> V
      } _4
    ] :> _4
;

syntax
  termBase ::= ...
    { [ "pack" typeIde_1 "<:" type_2 "=" type_3 "as" type_4
        "with" term_5 "end" ]
          => fun(V?) fun(f:All(_1<:_2)_4->V) f(:_3)(_5)
      [ "open" term_1 "as" typeIde_2 "<:" type_3 termIde_4 ":" type_5
        "in" term_6 "end" ]
          => _1(fun(_2<:_3)fun(_4:_5)_6)}
;
```

```
(* Example:

  load Bool; load Product;

  Let Spec = Some(X<:Bool) X*{X->Bool};

  let impl: Spec =
    pack X<:Bool=True as X*{X->Bool}
    with tt, fun(x:True)true end;

  open impl as X<:Bool p:X*{X->Bool}
  in snd(p)(fst(p)) end;

Note: trying to extract fst(p) rightfully causes a type-inference
rank-check, which would not be captured by the normal first-order
unification algorithm.
*)
```

Untyped λ-terms

This is the file ‘`Scott.fsub`’. It uses recursive types to encode the untyped λ-calculus.

```
module Scott;
(* Defines
    V = V->V
    \x e            untyped lambda
    e.e             untyped application
    i,k,s,y: V      the usual combinators
*)

Let V = Rec(V) V->V;

syntax
  termBase ::= ...
    ["\\" termIde_1 term_2]
      => fold(:V)(fun(_1:V)_2)
  termOper ::= ... *_1
    ["." termAppl_2]
      => unfold(_1)(_2);

let i = \x x
    k = \x \y x
    s = \x \y \z {x.z}.{y.z}
    y = \f {\x f.{x.x}}.{\x f.{x.x}};

(* Note: the evaluator is eager; k.i.{y.i} will diverge. To fix this, use:

module Scott
import Unit;

Let V = Rec(V) {Unit->V}->V;

syntax
  termBase ::= ...
    { ["@" termIde_1]
        => _1(unit)
      ["\\" termIde_1 term_2]
        => fold(:V)(fun(_1:Unit->V)_2) }
  termOper ::= ... *_1
    ["." termAppl_2]
        => unfold(_1)(fun(u:Unit)_2);

let i = \x @x;
let k = \x \y @x;
let s = \x \y \z {@x.@z}.{@y.@z};
let y = \f {\x @f.{@x.@x}}.{\x @f.{@x.@x}};

k.i.{y.i};

*)
```

Appendix B. Lexicon

The ASCII characters are divided into the following classes:

Blank	*HT LF FF CR SP*
Reserved	" ' ~
Delimiter	( ) , . ; [ ] _ { } ? !
Special	# $ % & * + - / : < = > @ \ ^ |
Digit	0 1 2 3 4 5 6 7 8 9
Letter	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
Illegal	all the others

Moreover:

- a StringChar is either
  - any single character that is not an Illegal character or one of ‘'’, ‘"’, ‘\’.
  - one of the pairs of characters ‘\'’, ‘\"’, ‘\\’.
- a Comment is, recursively, a sequence of non-Illegal characters and comments enclosed between ‘(*’ and ‘*)’.

From these, the following *lexemes* are formed:

Space	a sequence of Blanks and Comments.
AlphaNum	a sequence of Letters and Digits starting with a Letter.
Symbol	a sequence of Specials.
Char	a single StringChar enclosed between two ‘'’.
String	a sequence of StringChars enclosed between two ‘"’.
Int	a sequence of Digits, possibly preceded by a single minus sign ‘~’.
Delimiter	a single Delimiter character.

A stream of characters is split into lexemes by always extracting the longest prefix that is a lexeme. Note that Delimiters do not stick to each other or to other tokens even when they are not separated by Space, but some care must be taken so that Symbols are not inadvertently merged.

A *token* is either a Char, String, Int, Delimiter, Identifier, or Keyword. Once a stream of characters has been split into lexemes, *tokens* are extracted as follows.

- Space lexemes do not produce tokens.
- Char, String, Int, and Delimiter lexemes are also tokens.
- AlphaNum and Symbol lexemes are Identifier tokens, except when they have been explicitly declared to be *keywords*, in which case they are Keyword tokens.

Appendix C. Syntax

The predefined keywords are:

For grammar definitions:

`char end ide in int string syntax ::= => :> * =`

For F-sub proper:

`All Let Rec Top fold fun judge let rec top unfold : <: -> = |-`

- The grammar of phrases is as follows.

```
phrase ::=                         (* public *)
    { ";"
     [ "Let" typeBinding ";" ]
     [ "let" termBinding ";" ]
     [ ":" type ";" ]
     [ term ";" ]
     [ synDecl ";" ]
     [ "judge" { [ "env" env]
                 [ "type" env "|-" type ]
                 [ "subtype" env "|-" type "<:" type ]
                 [ "term" env "|-" term ":" type ]
                } ";" ]
     [ "reload" { ide string } ";" ]
     [ "restore" { ide [] } ";" ] [ { "save" "establish" "load" } ide ";" ]
     [ "module" ide { "import" ideList } ";" ]
     [ "do" { [ ide { ide [] }] [] } ";" ]
    }

ideList ::=
    { [ ide ideList ] [] }

typeBinding ::=
    { [ ide { [ "<:" type ] [] } "=" type typeBinding ] [] }

termBinding ::=
    { [ ide { [ ":" type ] [] } "=" term termBinding ] [] }

env ::=
    { [ ide { [ "<:" type ] [ ":" type ] } ] [] }
```

- The grammar of types and terms is as follows.

```
pvar ::=
    [ "_" int ]

binder ::=
    { ide pvar }
```

```
type ::=                        (* public *)
    [ typeOper { [ "->" type [] ] } ]

typeOper ::=                    (* public, hook for client infixes *)
    ( typeBase *_1 {} )

typeIde ::=                     (* public *)
    ide

typeBase ::=                    (* public *)
    { typeIde  pvar  "Top"
     [ "All" "(" binder { "?" [] } { [ "<:" type] [] } ")" type ]
     [ "Rec" "(" ide ")" type ]
     [ "{" type "}" ] }

term ::=                        (* public *)
    termOper

termOper ::=                    (* public, hook for client infixes *)
    (termAppl *_1 {})

termAppl ::=                    (* public *)
    ( termBase *_1
     { [ "(" { [ ":" type] term } ")" ]
       "!" } )                  (* "!" must follow an identifier or keyword *)

termIde ::=                     (* public *)
    ide

termBase ::=                    (* public *)
    { termIde  pvar "top"
     [ "fun" "(" binder { [ ":" type ] [ "<:" type ] [ "?" { [ "<:" type ] [] } ] [] } ")" term ]
     [ "fold" "(" ":" type ")" "(" term ")" ]
     [ "unfold" "(" term ")" ]
     [ "rec" "(" ide ":" type ")" term ]
     [ "{" term "}" ]
     synTerm }
```

- The grammar of syntax extensions is as follows. Note that the grammar for synTerm cannot be written down precisely in this notation.

```
synDecl ::=
    [ "syntax" { "toplevel" [] } grammar ]

synTerm ::=
    [ "syntax" grammar "in" ... "end" ]

grammar ::=
    clauseSeq

clauseSeq ::=
    [ ide "::=" extends gramExp { clauseSeq [] } ]
```

```
extends ::=
    { [ "." "." "." { [ "*" { pvar [] } [] } [] }

gramExp ::=
    [ gramExpBase
      { [ "=>" term ]
        [ ":>" type ]
        pvar
        [] } ]

gramExpBase ::=
    { ide string "ide" "int" "char" "string"
      [ "[" gramExpList "]" ]
      [ "{" gramExpList "}" ]
      [ "(" gramExp { [ "*" { pvar [] } gramExp ] [] } ")" ] }

gramExpList ::=
    { [ gramExp gramExpList ] [] }
```

Appendix D. Typing rules

These are the typing rules of F-sub, as described in [Cardelli, *et al.* 1991].

Environments

(Env $\emptyset$)

$$\frac{}{E \vdash \emptyset\ \text{env}}$$

(Env $\mathtt{x}$)

$$\frac{E \vdash \mathtt{A}\ \text{type} \qquad \mathtt{x} \notin \text{dom}(E)}{\vdash E, \mathtt{x} : \mathtt{A}\ \text{env}}$$

(Env $\mathtt{X}$)

$$\frac{E \vdash \mathtt{A}\ \text{type} \qquad \mathtt{X} \notin \text{dom}(E)}{\vdash E, \mathtt{X} <: \mathtt{A}\ \text{env}}$$

Types

(Type $\mathtt{X}$)

$$\frac{\vdash E, \mathtt{X} <: \mathtt{A}, E'\ \text{env}}{E, \mathtt{X} <: \mathtt{A}, E' \vdash \mathtt{X}\ \text{type}}$$

(Type $\mathtt{Top}$)

$$\frac{\vdash E\ \text{env}}{E \vdash \mathtt{Top}\ \text{type}}$$

(Type $->$)

$$\frac{E \vdash \mathtt{A}\ \text{type} \qquad E \vdash \mathtt{B}\ \text{type}}{E \vdash \mathtt{A} -> \mathtt{B}\ \text{type}}$$

(Type $\mathtt{All}$)

$$\frac{E, \mathtt{X} <: \mathtt{A} \vdash \mathtt{B}\ \text{type}}{E \vdash \mathtt{All}\,(\mathtt{X} <: \mathtt{A})\mathtt{B}\ \text{type}}$$

Subtypes

(Sub refl)

$$\frac{E \vdash \mathtt{A}\ \text{type}}{E \vdash \mathtt{A} <: \mathtt{A}}$$

(Sub trans)

$$\frac{E \vdash \mathtt{A} <: \mathtt{B} \qquad E \vdash \mathtt{B} <: \mathtt{C}}{E \vdash \mathtt{A} <: \mathtt{C}}$$

(Sub $\mathtt{X}$)

$$\frac{\vdash E, \mathtt{X} <: \mathtt{A}, E'\ \text{env}}{E, \mathtt{X} <: \mathtt{A}, E' \vdash \mathtt{X} <: \mathtt{A}}$$

(Sub $\mathtt{Top}$)

$$\frac{E \vdash \mathtt{A}\ \text{type}}{E \vdash \mathtt{A} <: \mathtt{Top}}$$

(Sub $->$)

$$\frac{E \vdash \mathtt{A'} <: \mathtt{A} \qquad E \vdash \mathtt{B} <: \mathtt{B'}}{E \vdash \mathtt{A} -> \mathtt{B} \quad <: \quad \mathtt{A'} -> \mathtt{B'}}$$

(Sub $\mathtt{All}$)

$$\frac{E \vdash \mathtt{A'} <: \mathtt{A} \qquad E, \mathtt{X} <: \mathtt{A'} \vdash \mathtt{B} <: \mathtt{B'}}{E \vdash \mathtt{All}\,(\mathtt{X} <: \mathtt{A})\mathtt{B} \quad <: \quad \mathtt{All}\,(\mathtt{X} <: \mathtt{A'})\mathtt{B'}}$$

Terms

(Subsumption)

$$\frac{E \vdash \mathtt{a} : \mathtt{A} \qquad E \vdash \mathtt{A} <: \mathtt{B}}{E \vdash \mathtt{a} : \mathtt{B}}$$

(Term $\mathtt{x}$)

$$\frac{\vdash E, \mathtt{x} : \mathtt{A}, E'\ \text{env}}{E, \mathtt{x} : \mathtt{A}, E' \vdash \mathtt{x} : \mathtt{A}}$$

(Term $\mathtt{top}$)

$$\frac{\vdash E\ \text{env}}{E \vdash \mathtt{top} : \mathtt{Top}}$$

(Term fun)

$$\frac{E, \mathtt{x} : \mathtt{A} \vdash \mathtt{b} : \mathtt{B}}{E \vdash \mathtt{fun}\,(\mathtt{x} : \mathtt{A})\mathtt{b} \quad : \quad \mathtt{A} -> \mathtt{B}}$$

(Term appl)

$$\frac{E \vdash \mathtt{b} : \mathtt{A} -> \mathtt{B} \qquad E \vdash \mathtt{a} : \mathtt{A}}{E \vdash \mathtt{b}(\mathtt{a}) \quad : \quad \mathtt{B}}$$

(Term fun2)

$$\frac{E, \mathtt{X} <: \mathtt{A} \vdash \mathtt{b} : \mathtt{B}}{E \vdash \mathtt{fun}\,(\mathtt{X} <: \mathtt{A})\mathtt{b} \quad : \quad \mathtt{All}\,(\mathtt{X} <: \mathtt{A})\mathtt{B}}$$

(Term appl2)

$$\frac{E \vdash \mathtt{b} : \mathtt{All}\,(\mathtt{X} <: \mathtt{A})\mathtt{B} \qquad E \vdash \mathtt{A'} <: \mathtt{A}}{E \vdash \mathtt{b}(:\mathtt{A'}) \quad : \quad \mathtt{B}\{\mathtt{X} \leftarrow \mathtt{A'}\}}$$

In preparation for the typing algorithms, these are the same type rules expressed with de Bruijn indices. The notation $\in A$ stands for either $:A$ or $<:A$. *Lifting* is $A\uparrow_i$, and *substitution* is $B\{i \leftarrow A\}$; the latter is to be invoked as $B\{1 \leftarrow A\}$.

$$A\uparrow_i = A\uparrow_i^0 \;; \qquad n\uparrow_i^j = n \;\; (n \le j) \;; \qquad n\uparrow_i^j = n + i \;\; (n > j) \;; \qquad \mathtt{Top}\uparrow_i^j = \mathtt{Top}$$

$$(A -> B)\uparrow_i^j = A\uparrow_i^j -> B\uparrow_i^{j+1} \;; \qquad (\mathtt{All}\,(<:A)B)\uparrow_i^j = \mathtt{All}\,(<:A\uparrow_i^j)B\uparrow_i^{j+1}$$

$$n\{i \leftarrow C\} = n \;\; (n < i) \;; \qquad n\{n \leftarrow C\} = C\uparrow_{n-1} \;; \qquad n\{i \leftarrow C\} = n - 1 \;\; (n > i) \;; \qquad \mathtt{Top}\{i \leftarrow C\} = \mathtt{Top}$$

$$(A -> B)\{i \leftarrow C\} = A\{i \leftarrow C\} -> B\{i+1 \leftarrow C\} \;; \qquad (\mathtt{All}\,(<:A)B)\{i \leftarrow C\} = \mathtt{All}\,(<:A\{i \leftarrow C\})B\{i+1 \leftarrow C\}$$

Environments

(Env ∅) $$\frac{}{E \vdash \emptyset \ \text{env}}$$

(Env x) $$\frac{E \vdash A \ \text{type}}{\vdash E,:A \ \text{env}}$$

(Env X) $$\frac{E \vdash A \ \text{type}}{\vdash E,<:A \ \text{env}}$$

Types

(Type X) $$\frac{\vdash E,<:A,\in A_{n-1},\ldots,\in A_1 \ \text{env}}{E,<:A,\in A_{n-1},\ldots,\in A_1 \vdash n \ \text{type}}$$

(Type `Top`*)* $$\frac{\vdash E \ \text{env}}{E \vdash \mathtt{Top} \ \text{type}}$$

(Type −>) $$\frac{E,:A \vdash B \ \text{type}}{E \vdash A -> B \ \text{type}}$$

(Type `All`*)* $$\frac{E,<:A \vdash B \ \text{type}}{E \vdash \mathtt{All}\,(<:A)B \ \text{type}}$$

Subtypes

(Sub refl) $$\frac{E \vdash A \ \text{type}}{E \vdash A <: A}$$

(Sub trans) $$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X) $$\frac{\vdash E,<:A,\in A_{n-1},\ldots,\in A_1 \ \text{env}}{E,<:A,\in A_{n-1},\ldots,\in A_1 \vdash n <: A\uparrow_n}$$

(Sub `Top`*)* $$\frac{E \vdash A \ \text{type}}{E \vdash A <: \mathtt{Top}}$$

(Sub −>) $$\frac{E \vdash A' <: A \qquad E,:A' \vdash B <: B'}{E \vdash A -> B \;\; <: \;\; A' -> B'}$$

(Sub `All`*)* $$\frac{E \vdash A' <: A \qquad E,<:A' \vdash B <: B'}{E \vdash \mathtt{All}\,(<:A)B \;\; <: \;\; \mathtt{All}\,(<:A')B'}$$

Terms

(Subsumption) $$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Term x*)* $$\frac{\vdash E,:A,\in A_{n-1},\ldots,\in A_1 \ \text{env}}{E,:A,\in A_{n-1},\ldots,\in A_1 \vdash n : A\uparrow_n}$$

(Term `top`*)* $$\frac{\vdash E \ \text{env}}{E \vdash \mathtt{top} : \mathtt{Top}}$$

(Term fun) $$\frac{E,:A \vdash b : B}{E \vdash \mathtt{fun}\,(:A)b \;\; : \;\; A -> B}$$

(Term appl) $$\frac{E \vdash b : A -> B \qquad E \vdash a : A}{E \vdash b(a) \;\; : \;\; B}$$

(Term fun2) $$\frac{E,<:A \vdash b : B}{E \vdash \mathtt{fun}\,(<:A)b \;\; : \;\; \mathtt{All}\,(<:A)B}$$

(Term appl2) $$\frac{E \vdash b : \mathtt{All}\,(<:A)B \qquad E \vdash A' <: A}{E \vdash b(:A') \;\; : \;\; B\{1 \leftarrow A'\}}$$

Appendix E. Typing algorithm

The parsing phase eliminates all the syntax extensions. A scoping phase converts variables to de Bruijn indices, and checks that all variables are properly bound. (Variables in type contexts should be bound by type binders, while variables in term contexts should be bound by term binders.) The scoping phase also expands all the top-level type definitions. Therefore, only the following data structures have to be considered for typechecking (where $n \geq 1$):

PreType

$$S,T = \ n \mid \texttt{Top} \mid S\texttt{->}T \mid \texttt{All(<:}S\texttt{)}T$$

PreTerm

$$a,b = \ n \mid \texttt{top} \mid \texttt{fun(:}S\texttt{)}b \mid b\texttt{(}a\texttt{)} \mid \texttt{fun(<:}S\texttt{)}b \mid b\texttt{(:}S\texttt{)}$$

Env

$$E = \ \emptyset \mid E\texttt{,<:}A \mid E\texttt{,:}A$$

Type

$$A,B = \ n \mid \texttt{Top} \mid A\texttt{->}B \mid \texttt{All(<:}A\texttt{)}B$$

The following algorithms are expressed in the form of deterministic labeled transition systems [Plotkin 1981]. (They can be read much as Prolog programs.) Each kind of “arrow” defines a (functional) relation. The name of the relation is on top of the arrow, the main parameters are on the left, additional parameters are below, and results are on the right. The main parameters are, by convention, the ones subject to structural induction. The signature of each relation is given in a box, and includes parameter names as comments; the notation *a*:*A*(=*b*) means that *b* is the default value of the parameter *a*, when that parameter is omitted.

There is a direct correspondence between each relation and a recursive procedure in the implementation code, and between each rule and a case branch in the implementation code.

To preserve all the internal invariants, we assumed that *type* and *term* are the top-level algorithms, and that they are started with an empty environment. Typechecking failure is (implicitly) represented as a “stuck” condition of the transition system.

What follows is the now well known sound and complete algorithm for F-sub [Curien, Ghelli 1991]. Giorgio Ghelli has shown that this algorithm diverges in some situations where it should fail, and Benjamin Pierce has further shown that the type system of Appendix D is undecidable [Pierce 1992].

Lift increases indices above a cutoff index.

$$\boxed{type : Type \frac{lift}{by : Int,\ cutoff : Int(=0)}\triangleright result : Type}$$

$$n \frac{lift}{i,j}\triangleright n \ (n \leq j) \qquad n \frac{lift}{i,j}\triangleright n+i \ (n > j) \qquad \texttt{Top}\frac{lift}{i,j}\triangleright\texttt{Top}$$

$$\frac{A\,\frac{\mathit{lift}}{i,j}\triangleright A' \qquad B\,\frac{\mathit{lift}}{i,j+1}\triangleright B'}{A\texttt{->}B\,\frac{\mathit{lift}}{i,j}\triangleright A'\texttt{->}B'} \qquad \frac{A\,\frac{\mathit{lift}}{i,j}\triangleright A' \qquad B\,\frac{\mathit{lift}}{i,j+1}\triangleright B'}{\texttt{All(<:}A\texttt{)}B\,\frac{\mathit{lift}}{i,j}\triangleright \texttt{All(<:}A'\texttt{)}B'}$$

Replace performs a substitution and lowers the *free* indices.

$$\boxed{\mathit{type} : \mathit{Type}\,\frac{\mathit{replace}}{\mathit{index : Int,\ with : Type,\ lift : Int(=0)}}\triangleright \mathit{result : Type}}$$

$$n\,\frac{\mathit{replace}}{i,C,l}\triangleright n\ (n<i) \qquad \frac{C\,\frac{\mathit{lift}}{l}\triangleright C'}{n\,\frac{\mathit{replace}}{n,C,l}\triangleright C'} \qquad n\,\frac{\mathit{replace}}{i,C,l}\triangleright n-1\ (n>i) \qquad \texttt{Top}\,\frac{\mathit{replace}}{i,C,l}\triangleright\texttt{Top}$$

$$\frac{A\,\frac{\mathit{replace}}{i,C,l}\triangleright A' \qquad B\,\frac{\mathit{replace}}{i+1,C,l+1}\triangleright B'}{A\texttt{->}B\,\frac{\mathit{replace}}{i,C,l}\triangleright A'\texttt{->}B'} \qquad \frac{A\,\frac{\mathit{replace}}{i,C,l}\triangleright A' \qquad B\,\frac{\mathit{replace}}{i+1,C,l+1}\triangleright B'}{\texttt{All(<:}A\texttt{)}B\,\frac{\mathit{replace}}{i,C,l}\triangleright \texttt{All(<:}A'\texttt{)}B'}$$

TypeIde extracts the bound of a type identifier from an environment.

$$\boxed{\mathit{env : Env}\,\frac{\mathit{typeIde}}{\mathit{index : Int,\ depth : Int(=index)}}\triangleright \mathit{result : Type}}$$

$$\frac{A\,\frac{\mathit{lift}}{d}\triangleright A'}{E,<:A\,\frac{\mathit{typeIde}}{1,d}\triangleright A'} \qquad \frac{E\,\frac{\mathit{typeIde}}{n,d}\triangleright B}{E,<:A\,\frac{\mathit{typeIde}}{n+1,d}\triangleright B} \qquad \frac{E\,\frac{\mathit{typeIde}}{n,d}\triangleright B}{E,:A\,\frac{\mathit{typeIde}}{n+1,d}\triangleright B}$$

TermIde extracts the type of a type identifier from an environment.

$$\boxed{\mathit{env : Env}\,\frac{\mathit{termIde}}{\mathit{index : Int,\ depth : Int(=index)}}\triangleright \mathit{result : Type}}$$

$$\frac{A\,\frac{\mathit{lift}}{d}\triangleright A'}{E,:A\,\frac{\mathit{termIde}}{1,d}\triangleright A'} \qquad \frac{E\,\frac{\mathit{termIde}}{n,d}\triangleright B}{E,<:A\,\frac{\mathit{termIde}}{n+1,d}\triangleright B} \qquad \frac{E\,\frac{\mathit{termIde}}{n,d}\triangleright B}{E,:A\,\frac{\mathit{termIde}}{n+1,d}\triangleright B}$$

ExposeArrow strips type variables until it finds an arrow type.

$$\boxed{\mathit{type} : \mathit{Type}\,\frac{\mathit{exposeArrow}}{\mathit{env : Env}}\triangleright \mathit{outType : Type}}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright A \qquad A \xrightarrow[E]{exposeArrow} \triangleright A'}{n \xrightarrow[E]{exposeArrow} \triangleright A'} \qquad A\text{-}{>}B \xrightarrow[E]{exposeArrow} \triangleright A\text{-}{>}B$$

ExposeAll strips type variables until it finds a forall type

$$\boxed{type : Type \xrightarrow[env : Env]{exposeAll} \triangleright outType : Type}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright A \qquad A \xrightarrow[E]{exposeAll} \triangleright A'}{n \xrightarrow[E]{exposeAll} \triangleright A'} \qquad \mathtt{All}(<: A)B \xrightarrow[E]{exposeAll} \triangleright \mathtt{All}(<: A)B$$

Sub tests subtyping between two types.

$$\boxed{small, big : Type \xrightarrow[env : Env]{sub} \triangleright result : \{\mathrm{ok}\}}$$

$$A, \mathtt{Top} \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad n, n \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad \frac{E \xrightarrow[n]{typeIde} \triangleright A \qquad A, B \xrightarrow[E]{sub} \triangleright \mathrm{ok}}{n, B \xrightarrow[E]{sub} \triangleright \mathrm{ok}} \ (B \neq n,\ B \neq \mathtt{Top})$$

$$\frac{A', A \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad B, B' \xrightarrow[E,: A']{sub} \triangleright \mathrm{ok}}{A\text{-}{>}B, A'\text{-}{>}B' \xrightarrow[E]{sub} \triangleright \mathrm{ok}} \qquad \frac{A', A \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad B, B' \xrightarrow[E, <: A']{sub} \triangleright \mathrm{ok}}{\mathtt{All}(<: A)B, \mathtt{All}(<: A')B' \xrightarrow[E]{sub} \triangleright \mathrm{ok}}$$

Type checks that a pretype is well-formed and returns the corresponding type.

$$\boxed{pre : PreType \xrightarrow[env : Env(=\emptyset)]{type} \triangleright result : Type}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright A}{n \xrightarrow[E]{type} \triangleright n} \qquad \mathtt{Top} \xrightarrow[E]{type} \triangleright \mathtt{Top} \qquad \frac{S \xrightarrow[E]{type} \triangleright A \qquad T \xrightarrow[E,: A]{type} \triangleright B}{S\text{-}{>}T \xrightarrow[E]{type} \triangleright A\text{-}{>}B} \qquad \frac{S \xrightarrow[E]{type} \triangleright A \qquad T \xrightarrow[E, <: A]{type} \triangleright B}{\mathtt{All}(<: S)T \xrightarrow[E]{type} \triangleright \mathtt{All}(<: A)B}$$

Term checks that a preterm is well-typed and returns its type.

$$\boxed{pre : PreTerm \xrightarrow[env : Env(=\emptyset)]{term} \triangleright result : Type}$$

$$\frac{E \xrightarrow[n]{termIde} \triangleright A}{n \xrightarrow[E]{term} \triangleright A} \qquad \mathtt{top} \xrightarrow[E]{term} \triangleright \mathtt{Top} \qquad \frac{S \xrightarrow[E]{type} \triangleright A \qquad b \xrightarrow[E,:A]{term} \triangleright B}{\mathtt{fun(:}S\mathtt{)}b \xrightarrow[E]{term} \triangleright A -> B} \qquad \frac{S \xrightarrow[E]{type} \triangleright A \qquad b \xrightarrow[E,<:A]{term} \triangleright B}{\mathtt{fun(<:}S\mathtt{)}b \xrightarrow[E]{term} \triangleright \mathtt{All(<:}A\mathtt{)}B}$$

$$\frac{b \xrightarrow[E]{term} \triangleright C \qquad C \xrightarrow[E]{exposeArrow} \triangleright A' -> B \qquad a \xrightarrow[E]{term} \triangleright A \qquad A, A' \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad B \xrightarrow[1, A]{replace} \triangleright B'}{b(a) \xrightarrow[E]{term} \triangleright B'}$$

$$\frac{b \xrightarrow[E]{term} \triangleright C \qquad C \xrightarrow[E]{exposeAll} \triangleright \mathtt{All(<:}A'\mathtt{)}B \qquad S \xrightarrow[E]{type} \triangleright A \qquad A, A' \xrightarrow[E]{sub} \triangleright \mathrm{ok} \qquad B \xrightarrow[1, A]{replace} \triangleright B'}{b(:S) \xrightarrow[E]{term} \triangleright B'}$$

Appendix F. Typing algorithm with argument synthesis

We now extend the typing algorithm of Appendix E with type inference. The inference mechanism is based syntactically on [Pollack 1990], and algorithmically on [Miller (to appear)].

The necessary data structures are as follows, where $q \equiv ?$ or q is empty.

PreType

$S, T \; = \; n \mid \mathtt{Top} \mid S\texttt{->}T \mid \mathtt{All}\,(q\texttt{<:}S)\,T$

PreTerm

$a, b \; = \; n \mid n! \mid \mathtt{top} \mid \mathtt{fun}\,(\texttt{:}S)\,b \mid b(a) \mid \mathtt{fun}\,(q\texttt{<:}S)\,b \mid b(\texttt{:}S)$

Env

$E \; = \; \emptyset \mid E,\texttt{<:}A \mid E,\texttt{:}A$

Type

$A, B \; = \; \alpha \mid n \mid \mathtt{Top} \mid A\texttt{->}B \mid \mathtt{All}\,(q\texttt{<:}A)\,B$

Subst

$\sigma \; = \; \emptyset \mid \alpha_r, \sigma \mid \alpha \leftarrow A, \sigma \mid \sigma\Uparrow \mid \sigma\Downarrow$

A substitution σ binds unification variables that may occur in terms, types and environments. An *instantiated* variable appears as $\alpha \leftarrow A$ in the substitution. A *non-instatiated* variable appears as α_r in the substitution, where the *rank r* is an index into an environment. Rank 1 points to the right of the rightmost component of the environment; rank 2 points between the rightmost component and the one to its left, and so on. This information encodes a *mixed prefix* [Miller (to appear)]: universally quantified variables are represented by de Bruijn indices into the environments, while existentially quantified (unification) variables have ranks pointing between components of the environment. (Therefore, the order of universal quantifiers matters, but the order of contiguous existential quantifiers does not.)

The operations $\sigma\Uparrow$ and $\sigma\Downarrow$ shift all the (free) de Bruijn indices and ranks in σ by +1 or -1.

Before describing the algorithm, we give some properties of substitutions. Here is how the substitution shifts $\sigma\Uparrow$ and $\sigma\Downarrow$ are normalized away, and how a normalized substitution σ is applied to a type A (via $A\{\sigma\}^0_0$). The order of occurrence of variables in a normalized substitution is not important.

$$\sigma\Uparrow = \sigma\uparrow_1 \; ; \qquad \sigma\Downarrow = \sigma\uparrow_{-1} \; ; \qquad \emptyset\uparrow_i = \emptyset \; ; \qquad (\alpha \leftarrow A, \sigma)\uparrow_i = \alpha \leftarrow A\uparrow_i, \sigma\uparrow_i \; ; \qquad (\alpha_r, \sigma)\uparrow_i = \alpha_{r+i}, \sigma\uparrow_i$$

$$A\uparrow_i = A\uparrow^0_i \; ; \qquad \alpha\uparrow^j_i = \alpha \; ; \qquad n\uparrow^j_i = n \;\; (n \le j) \; ; \qquad n\uparrow^j_i = n + i \;\; (n > j) \; ; \qquad \mathtt{Top}\uparrow^j_i = \mathtt{Top}$$

$$(A\texttt{->}B)\uparrow^j_i = A\uparrow^j_i\texttt{->}B\uparrow^{j+1}_i \; ; \qquad (\mathtt{All}\,(q\texttt{<:}A)B)\uparrow^j_i = \mathtt{All}\,(q\texttt{<:}A\uparrow^j_i)B\uparrow^{j+1}_i$$

$$\alpha\{\alpha_r, \sigma\}^j_i = \alpha \; ; \qquad \alpha\{\alpha \leftarrow A, \sigma\}^j_i = A\{\sigma\}^0_j \; ; \qquad n\{\sigma\}^j_i = n \;\; (n \le j) \; ; \qquad n\{\sigma\}^j_i = n + i \;\; (n > j)$$

$$\mathtt{Top}\{\sigma\}^j_i = \mathtt{Top} \; ; \quad (A\texttt{->}B)\{\sigma\}^j_i = A\{\sigma\}^j_i\texttt{->}B\{\sigma\}^{j+1}_i \; ; \quad (\mathtt{All}\,(q\texttt{<:}A)B)\{\sigma\}^j_i = \mathtt{All}\,(q\texttt{<:}A\{\sigma\}^j_i)B\{\sigma\}^{j+1}_i$$

A substitution is applied to a judgment, for example a typing judgment, as follows. Indices decrease while moving into the environment, but all indices remain positive because of rank restrictions.

$$(E \vdash A \text{ type})\{\sigma\}^j_i = E\{\sigma\}^j_i \vdash A\{\sigma\}^j_i \text{ type}$$

$$\emptyset\{\sigma\}^j_i = \emptyset \; ; \qquad (E,\texttt{:}A)\{\sigma\}^j_i = E\{\sigma\}^j_{i\text{-}1},\texttt{:}A\{\sigma\}^j_{i\text{-}1} \; ; \qquad (E,\texttt{<:}A)\{\sigma\}^j_i = E\{\sigma\}^j_{i\text{-}1},\texttt{<:}A\{\sigma\}^j_{i\text{-}1}$$

Here is how an index-1-replacement operation $\mathtt{A}\{1 \leftarrow \mathtt{C}\}$ is applied to a type $\mathtt{A}$ containing unification variables. This operation occurs in the context of some substitution σ; the case $\alpha\{i \leftarrow \mathtt{C}\} = \alpha$ is justified by rank restrictions that ensure that whatever α is instantiated to, it cannot depend on *i*.

$$\alpha\{i \leftarrow \mathtt{C}\} = \alpha\,; \quad \mathtt{n}\{i \leftarrow \mathtt{C}\} = \mathtt{n} \;\; (\mathtt{n} < i)\,; \quad \mathtt{n}\{\mathtt{n} \leftarrow \mathtt{C}\} = \mathtt{C}\!\uparrow_{n\text{-}1}$$

$$\mathtt{n}\{i \leftarrow \mathtt{C}\} = \mathtt{n} - 1 \;\; (\mathtt{n} > i)\,; \quad \mathtt{Top}\{i \leftarrow \mathtt{C}\} = \mathtt{Top}$$

$$(\mathtt{A}{-}{>}\,\mathtt{B})\{i \leftarrow \mathtt{C}\} = \mathtt{A}\{i \leftarrow \mathtt{C}\}{-}{>}\,\mathtt{B}\{i+1 \leftarrow \mathtt{C}\}\,; \quad (\mathtt{All}\,(\mathtt{q} <: \mathtt{A})\mathtt{B})\{i \leftarrow \mathtt{C}\} = \mathtt{All}\,(\mathtt{q} <: \mathtt{A}\{i \leftarrow \mathtt{C}\})\mathtt{B}\{i+1 \leftarrow \mathtt{C}\}$$

The notation $\sigma \backslash \alpha$ indicates removing $\alpha{\leftarrow}\mathtt{A}$ or α_r from σ.

Lift increases indices above a cutoff index.

$$\boxed{\mathit{type : Type}\ \frac{\mathit{lift}}{\mathit{by : Int, cutoff : Int(=0)}}\triangleright \mathit{result : Type}}$$

$$\alpha\frac{\mathit{lift}}{i,j}\triangleright\alpha \qquad \mathtt{n}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{n}\;(n \le j) \qquad \mathtt{n}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{n}+1\;(n > j) \qquad \mathtt{Top}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{Top}$$

$$\dfrac{\mathtt{A}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{A'} \qquad \mathtt{B}\frac{\mathit{lift}}{i,j+1}\triangleright\mathtt{B'}}{\mathtt{A}{-}{>}\,\mathtt{B}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{A'}{-}{>}\,\mathtt{B'}} \qquad \dfrac{\mathtt{A}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{A'} \qquad \mathtt{B}\frac{\mathit{lift}}{i,j+1}\triangleright\mathtt{B'}}{\mathtt{All}\,(\mathtt{q} <: \mathtt{A})\mathtt{B}\frac{\mathit{lift}}{i,j}\triangleright\mathtt{All}\,(\mathtt{q} <: \mathtt{A'})\mathtt{B'}}$$

Retrieve retrieves the instance or rank of a type variable from a substitution.

$$\boxed{\mathit{subst : Subst}\ \frac{\mathit{retrieve}}{\mathit{lift : Int(=0),\ var : TypeVar}}\triangleright \mathit{instance : Type} \cup \mathit{rank : Int}}$$

$$\dfrac{\mathtt{A}\frac{\mathit{lift}}{i}\triangleright\mathtt{A'}}{\alpha \leftarrow \mathtt{A}, \sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A'}} \qquad \dfrac{\sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}}{\beta \leftarrow \mathtt{B}, \sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}}\;(\beta \ne \alpha)$$

$$\alpha_r, \sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright r+i \qquad \dfrac{\sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}}{\beta_r, \sigma\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}}\;(\beta \ne \alpha) \qquad \dfrac{\sigma\frac{\mathit{retrieve}}{i+1,\alpha}\triangleright\mathtt{A}}{\sigma\Uparrow\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}} \qquad \dfrac{\sigma\frac{\mathit{retrieve}}{i-1,\alpha}\triangleright\mathtt{A}}{\sigma\Downarrow\frac{\mathit{retrieve}}{i,\alpha}\triangleright\mathtt{A}}$$

Replace performs a substitution with an index shift of -1 on “free indices”. A type variable shall not depend on the replacement index.

$$\boxed{\mathit{type : Type}\ \frac{\mathit{replace}}{\mathit{index : Int,\ with : Type}}\triangleright \mathit{result : Type}}$$

$$\alpha \xrightarrow[i,C]{\mathit{replace}} \alpha \qquad n \xrightarrow[i,C]{\mathit{replace}} n\ (n < i) \qquad \dfrac{C \xrightarrow[n-1]{\mathit{lift}} C'}{n \xrightarrow[n,C]{\mathit{replace}} C'} \qquad n \xrightarrow[i,C]{\mathit{replace}} n-1\ (n > i) \qquad \mathtt{Top} \xrightarrow[i,C]{\mathit{replace}} \mathtt{Top}$$

$$\dfrac{A \xrightarrow[i,C]{\mathit{replace}} A' \qquad B \xrightarrow[i+1,C]{\mathit{replace}} B'}{A -> B \xrightarrow[i,C]{\mathit{replace}} A' -> B'} \qquad \dfrac{A \xrightarrow[i,C]{\mathit{replace}} A' \qquad B \xrightarrow[i+1,C]{\mathit{replace}} B'}{\mathtt{All}\,(q <: A)B \xrightarrow[i,C]{\mathit{replace}} \mathtt{All}\,(q <: A')B'}$$

Strip expands question-mark quantifiers and introduces ranked variables in substitutions.

$$\boxed{\mathit{type : Type} \xrightarrow[\mathit{subst : Subst}]{\mathit{strip}} \mathit{outType : Type,\ \ outSubst : Subst}}$$

$$\dfrac{\sigma \xrightarrow[\alpha]{\mathit{retrieve}} A \qquad A \xrightarrow[\sigma]{\mathit{strip}} A', \sigma'}{\alpha \xrightarrow[\sigma]{\mathit{strip}} A', \sigma'} \qquad \dfrac{\sigma \xrightarrow[\alpha]{\mathit{retrieve}} r}{\alpha \xrightarrow[\sigma]{\mathit{strip}} \alpha, \sigma} \qquad A \xrightarrow[\sigma]{\mathit{strip}} A, \sigma\ \ (A \neq \alpha,\ A \neq \mathtt{All}\,(?<:C)B)$$

$$\dfrac{B \xrightarrow[1,\alpha]{\mathit{replace}} B' \qquad B' \xrightarrow[\alpha_1,\sigma]{\mathit{strip}} B'', \sigma'}{\mathtt{All}\,(?<:\mathtt{Top})B \xrightarrow[\sigma]{\mathit{strip}} B'', \sigma'}\ (\alpha \text{ new in } \sigma) \qquad \dfrac{B \xrightarrow[1,A]{\mathit{replace}} B' \qquad B' \xrightarrow[\sigma]{\mathit{strip}} B'', \sigma'}{\mathtt{All}\,(?<:A)B \xrightarrow[\sigma]{\mathit{strip}} B'', \sigma'}\ (A \neq \mathtt{Top})$$

TypeIde extracts the bound of a type identifier from an environment.

$$\boxed{\mathit{env : Env} \xrightarrow[\mathit{index : Int,\ \ depth : Int(= index)}]{\mathit{typeIde}} \mathit{result : Type}}$$

$$\dfrac{A \xrightarrow[d]{\mathit{lift}} A'}{E, <: A \xrightarrow[1,d]{\mathit{typeIde}} A'} \qquad \dfrac{E \xrightarrow[n,d]{\mathit{typeIde}} B}{E, <: A \xrightarrow[n+1,d]{\mathit{typeIde}} B} \qquad \dfrac{E \xrightarrow[n,d]{\mathit{typeIde}} B}{E, : A \xrightarrow[n+1,d]{\mathit{typeIde}} B}$$

TermIde extracts the type of a type identifier from an environment.

$$\boxed{\mathit{env : Env} \xrightarrow[\mathit{index : Int,\ \ depth : Int(= index)}]{\mathit{termIde}} \mathit{result : Type}}$$

$$\dfrac{A \xrightarrow[d]{\mathit{lift}} A'}{E, : A \xrightarrow[1,d]{\mathit{termIde}} A'} \qquad \dfrac{E \xrightarrow[n,d]{\mathit{termIde}} B}{E, <: A \xrightarrow[n+1,d]{\mathit{termIde}} B} \qquad \dfrac{E \xrightarrow[n,d]{\mathit{termIde}} B}{E, : A \xrightarrow[n+1,d]{\mathit{termIde}} B}$$

ExposeArrow strips variables until it can find or generate an arrow type.

$$\boxed{type : Type \xrightarrow[env : Env,\ subst : Subst]{exposeArrow} \triangleright outType : Type,\ outSubst : Subst}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright \mathtt{A} \qquad \mathtt{A} \xrightarrow[E,\sigma]{exposeArrow} \triangleright \mathtt{A}', \sigma'}{n \xrightarrow[E,\sigma]{exposeArrow} \triangleright \mathtt{A}', \sigma'} \qquad \mathtt{A} -> \mathtt{B} \xrightarrow[E,\sigma]{exposeArrow} \triangleright \mathtt{A} -> \mathtt{B}, \sigma$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright \mathtt{A} \qquad \mathtt{A} \xrightarrow[E,\sigma]{exposeArrow} \triangleright \mathtt{A}', \sigma'}{\alpha \xrightarrow[E,\sigma]{exposeArrow} \triangleright \mathtt{A}', \sigma'}$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright r}{\alpha \xrightarrow[E,\sigma]{exposeArrow} \triangleright (\alpha' -> \alpha''), (\alpha \leftarrow (\alpha' -> \alpha''), \alpha'_r, \alpha''_r, (\sigma \setminus \alpha))} \ (\alpha', \alpha'' \text{ new in } \sigma)$$

ExposeForall strips variables until it can find a forall type or can generate a (non-`?`) forall type.

$$\boxed{type : Type \xrightarrow[env : Env,\ subst : Subst]{exposeAll} \triangleright outType : Type,\ outSubst : Subst}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright \mathtt{A} \qquad \mathtt{A} \xrightarrow[E,\sigma]{exposeAll} \triangleright \mathtt{A}', \sigma'}{n \xrightarrow[E,\sigma]{exposeAll} \triangleright \mathtt{A}', \sigma'} \qquad \mathtt{All}\,(q <: \mathtt{A})\mathtt{B} \xrightarrow[E,\sigma]{exposeAll} \triangleright \mathtt{All}\,(q <: \mathtt{A})\mathtt{B}, \sigma$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright \mathtt{A} \qquad \mathtt{A} \xrightarrow[E,\sigma]{exposeAll} \triangleright \mathtt{A}', \sigma'}{\alpha \xrightarrow[E,\sigma]{exposeAll} \triangleright \mathtt{A}', \sigma'}$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright r}{\alpha \xrightarrow[E,\sigma]{exposeAll} \triangleright (\mathtt{All}\,(<: \alpha')\alpha''), (\alpha \leftarrow (\mathtt{All}\,(<: \alpha')\alpha''), \alpha'_r, \alpha''_r, (\sigma \setminus \alpha))} \ (\alpha', \alpha'' \text{ new in } \sigma)$$

OccurCheck tests for circular instantiations and rank violations (variable captures).

$$\boxed{type : Type \xrightarrow[var : TypeVar,\ varRank : Int,\ subst : Subst,\ level : Int(=0)]{occurCheck} \triangleright result : Subst}$$

$$\dfrac{\sigma \frac{\mathit{retrieve}}{\beta}\triangleright s}{\beta \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \beta_r, (\sigma \setminus \beta)}\ (\alpha \neq \beta,\ s < r) \qquad \dfrac{\sigma \frac{\mathit{retrieve}}{\beta}\triangleright s}{\beta \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma}\ (\alpha \neq \beta,\ s \geq r)$$

$$\dfrac{\sigma \frac{\mathit{retrieve}}{\beta}\triangleright B \qquad B \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma'}{\beta \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma'}\ (\alpha \neq \beta) \qquad n \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma\ (\neg(n > i \wedge n < r)) \qquad \mathtt{Top} \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma$$

$$\dfrac{A \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma' \qquad B \frac{\mathit{occurCheck}}{\alpha, r+1, \sigma'\Uparrow, i+1}\triangleright \sigma''}{A -> B \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma''\Downarrow} \qquad \dfrac{A \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma' \qquad B \frac{\mathit{occurCheck}}{\alpha, r+1, \sigma'\Uparrow, i+1}\triangleright \sigma''}{\mathtt{All}\,(q <: A)B \frac{\mathit{occurCheck}}{\alpha, r, \sigma, i}\triangleright \sigma''\Downarrow}$$

Sub tests subtyping between two types.

$$\boxed{\mathit{small}, \mathit{big} : \mathit{Type} \frac{\mathit{sub}}{\mathit{env} : \mathit{Env},\ \mathit{substIn} : \mathit{Subst}}\triangleright \mathit{substOut} : \mathit{Subst}}$$

$$A, \mathtt{Top} \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma \qquad \alpha, \alpha \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma$$

$$\dfrac{\sigma \frac{\mathit{retrieve}}{\alpha}\triangleright A \qquad A, B \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma'}{\alpha, B \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma'}\ (B \neq \alpha,\ B \neq \mathtt{Top}) \qquad \dfrac{\sigma \frac{\mathit{retrieve}}{\beta}\triangleright B \qquad A, B \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma'}{A, \beta \frac{\mathit{sub}}{E, \sigma}\triangleright E, \sigma}\ (A \neq \beta)$$

$$\dfrac{\sigma \frac{\mathit{retrieve}}{\alpha}\triangleright r \qquad B \frac{\mathit{occurCheck}}{\alpha, r, \sigma}\triangleright \sigma'}{\alpha, B \frac{\mathit{sub}}{E, \sigma}\triangleright \alpha \leftarrow B, (\sigma' \setminus \alpha)}\ (B \neq \alpha,\ B \neq \mathtt{Top}) \qquad \dfrac{\sigma \frac{\mathit{retrieve}}{\beta}\triangleright r \qquad A \frac{\mathit{occurCheck}}{\beta, r, \sigma}\triangleright \sigma'}{A, \beta \frac{\mathit{sub}}{E, \sigma}\triangleright \beta \leftarrow A, (\sigma' \setminus \beta)}\ (A \neq \beta)$$

$$n, n \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma \qquad \dfrac{E \frac{\mathit{typeIde}}{n}\triangleright A \qquad A, B \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma'}{n, B \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma'}\ (B \neq \gamma,\ B \neq n,\ B \neq \mathtt{Top})$$

$$\dfrac{A', A \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma' \qquad B, B' \frac{\mathit{sub}}{E, : A', \sigma'\Uparrow}\triangleright \sigma''}{A -> B, A' -> B' \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma''\Downarrow} \qquad \dfrac{A', A \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma' \qquad B, B' \frac{\mathit{sub}}{E, <: A', \sigma'\Uparrow}\triangleright \sigma''}{\mathtt{All}\,(q <: A)B, \mathtt{All}\,(q <: A')B' \frac{\mathit{sub}}{E, \sigma}\triangleright \sigma''\Downarrow}$$

Type checks that a pretype is well-formed and returns the corresponding type.

$$\boxed{\mathit{pre : PreType}\,\frac{\mathit{type}}{\mathit{env : Env(=\emptyset)}}\triangleright \mathit{result : Type}}$$

$$\frac{E\,\frac{\mathit{typeIde}}{n}\triangleright A}{n\,\frac{\mathit{type}}{E}\triangleright n}\qquad \mathtt{Top}\,\frac{\mathit{type}}{E}\triangleright\mathtt{Top}\qquad \frac{S\,\frac{\mathit{type}}{E}\triangleright A \qquad T\,\frac{\mathit{type}}{E,:A}\triangleright B}{S\mathtt{->}T\,\frac{\mathit{type}}{E}\triangleright A\mathtt{->}B}\qquad \frac{S\,\frac{\mathit{type}}{E}\triangleright A \qquad T\,\frac{\mathit{type}}{E,<:A}\triangleright B}{\mathtt{All}\,(q<:S)T\,\frac{\mathit{type}}{E}\triangleright\mathtt{All}\,(q<:A)B}$$

Term checks that a preterm is well-typed and returns its type.

$$\boxed{\mathit{pre : PreTerm}\,\frac{\mathit{term}}{\mathit{env : Env(=\emptyset),\ substIn : Subst}}\triangleright \mathit{type : Type,\ substOut : Subst}}$$

$$\frac{E\,\frac{\mathit{termIde}}{n}\triangleright A \qquad A\,\frac{\mathit{strip}}{E,\sigma}\triangleright A',\sigma'}{n\,\frac{\mathit{term}}{E,\sigma}\triangleright A',\sigma'}\qquad \frac{E\,\frac{\mathit{termIde}}{n}\triangleright A}{n!\,\frac{\mathit{term}}{E,\sigma}\triangleright A,\sigma}\qquad \mathtt{top}\,\frac{\mathit{term}}{E,\sigma}\triangleright\mathtt{Top},\sigma$$

$$\frac{S\,\frac{\mathit{type}}{E}\triangleright A \qquad b\,\frac{\mathit{term}}{E,:A,\sigma\Uparrow}\triangleright B,\sigma'}{\mathtt{fun}\,(:S)b\,\frac{\mathit{term}}{E,\sigma}\triangleright A\mathtt{->}B,\sigma'\Downarrow}\qquad \frac{S\,\frac{\mathit{type}}{E}\triangleright A \qquad b\,\frac{\mathit{term}}{E,<:A,\sigma\Uparrow}\triangleright B,\sigma'}{\mathtt{fun}\,(q<:S)b\,\frac{\mathit{term}}{E,\sigma}\triangleright\mathtt{All}\,(q<:A)B,\sigma'\Downarrow}$$

$$\frac{b\,\frac{\mathit{term}}{E,\sigma}\triangleright C,\sigma' \qquad C\,\frac{\mathit{exposeArrow}}{\sigma'}\triangleright A'\mathtt{->}B,\sigma'' \qquad a\,\frac{\mathit{term}}{E,\sigma''}\triangleright A,\rho \qquad A,A'\,\frac{\mathit{sub}}{E,\rho}\triangleright\rho' \qquad B\,\frac{\mathit{replace}}{1,A}\triangleright B'}{b(a)\,\frac{\mathit{term}}{E,\sigma}\triangleright B',\rho'}$$

$$\frac{b\,\frac{\mathit{term}}{E,\sigma}\triangleright C,\sigma' \qquad C\,\frac{\mathit{exposeAll}}{\sigma'}\triangleright\mathtt{All}\,(q<:A')B,\rho \qquad S\,\frac{\mathit{type}}{E}\triangleright A \qquad A,A'\,\frac{\mathit{sub}}{E,\rho}\triangleright\rho' \qquad B\,\frac{\mathit{replace}}{1,A}\triangleright B'}{b(:S)\,\frac{\mathit{term}}{E,\sigma}\triangleright B',\rho'}$$

Appendix G. Recursion

As explained in section 7, two recursive types are equal only if the `Rec` bindings occur in the same positions. That is, unfolding a recursion does not produce, in general, an identical type. Since variables are replaced by de Bruijn indices, equality of recursive types is then simple identity.

Still, precisely because of de Bruijn indices, the subtyping test for recursive types is not trivial. The formal subtyping rule requires that the body of two recursive types be tested for inclusion under the assumption that the corresponding variables are included *in one direction*. But since the de Bruijn indices are identical in both types, they will match in *both directions*. For example, `Rec(X)X->Bool <: Rec(Y)Y->Top` should fail, while the de Bruijn version `Rec()1->Bool <: Rec()1->Top` would, naively, succeed. Hence, before testing the bodies we compute the *ties* between the recursion variables, which can be positive (covariant), negative (contravariant), or both. If the ties are only positive, we test the bodies for inclusion, otherwise we test the bodies for equality (inclusion in both directions). The ties are computed by mimicking a subtype test.

G.1. Typing Rules

Contractivity Relation ($A \succ X$)

$Y \succ X \Leftrightarrow Y \neq X$; $\texttt{Top} \succ X$; $(A -> B) \succ X$; $(\texttt{All}\,(Y <: A)B) \succ X$

$(\texttt{Rec}\,(Y)B) \succ X \Leftrightarrow B \succ X \wedge B \succ Y \wedge Y \neq X$

Types

(Type `Rec`*)*

$$\frac{E, X <: \texttt{Top} \vdash B \text{ type}}{E \vdash \texttt{Rec}\,(X)B \text{ type}}\;(B \succ X)$$

Subtypes

(Sub `Rec`*)*

$$\frac{E \vdash \texttt{Rec}\,(X)B \text{ type} \qquad E \vdash \texttt{Rec}\,(Y)C \text{ type} \qquad E, Y <: \texttt{Top}, X <: Y \vdash B <: C}{E \vdash \texttt{Rec}\,(X)B \;<:\; \texttt{Rec}\,(Y)C}$$

Terms

(Term `fold`*)*

$$\frac{E \vdash a : B\{X \leftarrow \texttt{Rec}\,(X)B\}}{E \vdash \texttt{fold}\,(:\texttt{Rec}\,(X)B)\;(a) \;:\; \texttt{Rec}\,(X)B}$$

(Term `unfold`*)*

$$\frac{E \vdash a : \texttt{Rec}\,(X)B}{E \vdash \texttt{unfold}\,(a) \;:\; B\{X \leftarrow \texttt{Rec}\,(X)B\}}$$

(Term `rec`*)*

$$\frac{E, x : A \vdash a : A}{E \vdash \texttt{rec}\,(x : A)a \;:\; A}$$

This is now the de Bruijn-index version.

$$(\texttt{Rec}\,()B){\uparrow}_i^j = \texttt{Rec}\,()B{\uparrow}_i^{j+1} \qquad\qquad (\texttt{Rec}\,()B)\{i \leftarrow C\} = \texttt{Rec}\,()B\{i+1 \leftarrow C\}$$

$$A{\succ} \Leftrightarrow A{\succ}1\,; \qquad n{\succ}m \Leftrightarrow n \neq m\,; \qquad \texttt{Top}{\succ}n\,; \qquad (A{-}{>}\,B){\succ}n\ ; \qquad (\texttt{All}\,(\texttt{<:}A)B){\succ}n$$

$$(\texttt{Rec}\,()B){\succ}n \Leftrightarrow B{\succ}n+1 \wedge B{\succ}1$$

Types

(*Type* `Rec`)

$$\frac{E, \texttt{<:}\,\texttt{Top} \vdash B \text{ type}}{E \vdash \texttt{Rec}\,()B \text{ type}}\,(B{\succ})$$

Subtypes

(*Sub* `Rec`)

$$\frac{E, \texttt{<:}\,\texttt{Top}, \texttt{<:}\,1 \vdash B{\uparrow}_1^1 \;\texttt{<:}\; C{\uparrow}_1^0}{E \vdash \texttt{Rec}\,()B \;\texttt{<:}\; \texttt{Rec}\,()C}\,(B{\succ}, C{\succ})$$

Terms

(*Term* `fold`)

$$\frac{E \vdash a : B\{1 \leftarrow \texttt{Rec}\,()B\}}{E \vdash \texttt{fold}\,(\texttt{:}\texttt{Rec}\,()B)\ (a)\ :\ \texttt{Rec}\,()B}$$

(*Term* `unfold`)

$$\frac{E \vdash a : \texttt{Rec}\,()B}{E \vdash \texttt{unfold}\,(a)\ :\ B\{1 \leftarrow \texttt{Rec}\,()B\}}$$

(*Term* `rec`)

$$\frac{E, \texttt{:}\,A \vdash a : A{\uparrow}_1}{E \vdash \texttt{rec}\,(\texttt{:}A)a\ :\ A}$$

G.2. Typing algorithm

PreType

$S, T \;=\; \ldots \;\mid\; \texttt{Rec()}\,T$

PreTerm

$a, b \;=\; \ldots \;\mid\; \texttt{fold(:}T\texttt{)(}a\texttt{)} \;\mid\; \texttt{unfold(}a\texttt{)} \;\mid\; \texttt{rec(:}S\texttt{)}\,b$

Type

$A, B \;=\; \ldots \;\mid\; \texttt{Rec()}\,B$

$$\frac{B \xrightarrow[i,\, j+1]{lift}\triangleright B'}{\texttt{Rec()}B \xrightarrow[i,\, j]{lift}\triangleright \texttt{Rec()}B'} \qquad \frac{B \xrightarrow[i+1,\, C,\, l+1]{replace}\triangleright B'}{\texttt{Rec()}B \xrightarrow[i,\, C,\, l]{replace}\triangleright \texttt{Rec()}B'} \qquad \frac{T \xrightarrow[E,<:\texttt{Top}]{type}\triangleright B \qquad B \xrightarrow{contracts}\triangleright \mathrm{ok}}{\texttt{Rec()}T \xrightarrow[E]{type}\triangleright \texttt{Rec()}B}$$

$$\frac{B, C \xrightarrow[E,<:\texttt{Top}]{ties}\triangleright \zeta \qquad B, C \xrightarrow[E,<:\texttt{Top}]{sub}\triangleright \mathrm{ok}}{\texttt{Rec()}B, \texttt{Rec()}C \xrightarrow[E]{sub}\triangleright \mathrm{ok}} \; (\zeta \subseteq \{\mathrm{pos}\})$$

$$\frac{B, C \xrightarrow[E,<:\texttt{Top}]{ties}\triangleright \zeta \qquad B, C \xrightarrow[E,<:\texttt{Top}]{sub}\triangleright \mathrm{ok} \qquad C, B \xrightarrow[E,<:\texttt{Top}]{sub}\triangleright \mathrm{ok}}{\texttt{Rec()}B, \texttt{Rec()}C \xrightarrow[E]{sub}\triangleright \mathrm{ok}} \; (\zeta \not\subseteq \{\mathrm{pos}\})$$

$$\frac{T \xrightarrow[E]{type}\triangleright \texttt{Rec()}B \qquad a \xrightarrow[E]{term}\triangleright A \qquad B \xrightarrow[1,\, \texttt{Rec()}B]{replace}\triangleright B' \qquad A, B' \xrightarrow[E]{sub}\triangleright \mathrm{ok}}{\texttt{fold(:}T\texttt{)}\;(a) \xrightarrow[E]{term}\triangleright \texttt{Rec()}B}$$

$$\frac{a \xrightarrow[E]{term}\triangleright A \qquad A \xrightarrow[E]{exposeRec}\triangleright \texttt{Rec()}B \qquad B \xrightarrow[1,\, \texttt{Rec()}B]{replace}\triangleright B'}{\texttt{unfold}(a) \xrightarrow[E]{term}\triangleright B'} \qquad \frac{S \xrightarrow[E]{type}\triangleright A \qquad a \xrightarrow[E,:\,A]{term}\triangleright A \qquad A \xrightarrow[1]{lift}\triangleright A' \qquad B, A' \xrightarrow[E,:\,A]{sub}\triangleright \mathrm{ok}}{\texttt{rec(:}S\texttt{)}\;(a) \xrightarrow[E]{term}\triangleright A}$$

Contracts tests whether a type is formally contractive in a variable.

$$\boxed{type : Type \xrightarrow[index\,:\,Int(=1)]{contracts}\triangleright result : \{\mathrm{ok}\}}$$

$$n \xrightarrow[m]{contracts}\triangleright \mathrm{ok}\; (n \neq m) \qquad \texttt{Top} \xrightarrow[m]{contracts}\triangleright \mathrm{ok} \qquad A\texttt{->}B \xrightarrow[m]{contracts}\triangleright \mathrm{ok} \qquad \texttt{All(<:}A\texttt{)}B \xrightarrow[m]{contracts}\triangleright \mathrm{ok}$$

$$\frac{B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{contracts}}_{1}\triangleright \mathrm{ok} \qquad B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{contracts}}_{m+1}\triangleright \mathrm{ok}}{\mathtt{Rec}\,()B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{contracts}}_{m}\triangleright \mathrm{ok}}$$

ExposeRec strips type variables until it finds a recursive type.

$$\boxed{\mathit{type : Type} \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{exposeRec}}_{\mathit{env : Env}}\triangleright \mathit{outType : Type}}$$

$$\frac{E \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{typeIde}}_{n}\triangleright A \qquad A \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{exposeRec}}_{E}\triangleright A'}{n \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{exposeRec}}_{E}\triangleright A'} \qquad \mathtt{Rec}\,()B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{exposeRec}}_{E}\triangleright \mathtt{Rec}\,()B$$

Ties computes the subtyping constraints between two recursive variables, by mimicking *sub*.

$$\boxed{\mathit{small, big : Type} \mathop{\rule{16em}{0.4pt}}\limits^{\mathit{ties}}_{\mathit{env : Env,\ index : Int(= 1),\ variance :} \{\mathrm{pos},\mathrm{neg}\}\mathit{(=} \mathrm{pos}\mathit{)}}\triangleright \mathit{result :} \mathcal{P}\{\mathrm{pos},\mathrm{neg}\}}$$

$$A,\mathtt{Top} \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \{\,\} \quad n,n \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,n,v}\triangleright \{v\} \quad n,n \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \{\,\}\ (i \neq n) \quad \frac{E \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{typeIde}}_{n}\triangleright A \qquad A,B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \zeta}{n,B \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \zeta}\ (B \neq n,\ B \neq \mathtt{Top})$$

$$\frac{A',A \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,\neg v}\triangleright \zeta \qquad B,B' \mathop{\rule{6em}{0.4pt}}\limits^{\mathit{ties}}_{E,:A',i+1,v}\triangleright \zeta'}{A\text{->}B, A'\text{->}B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \zeta \cup \zeta'} \qquad \frac{A',A \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,\neg v}\triangleright \zeta \qquad B,B' \mathop{\rule{6em}{0.4pt}}\limits^{\mathit{ties}}_{E,<:A',i+1,v}\triangleright \zeta'}{\mathtt{All}(<:A)B, \mathtt{All}(<:A')B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \zeta \cup \zeta'}$$

$$\frac{B,B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,1,\mathrm{pos}}\triangleright \zeta \qquad B,B' \mathop{\rule{6em}{0.4pt}}\limits^{\mathit{ties}}_{E,<:\mathtt{Top},i+1,v}\triangleright \zeta'}{\mathtt{Rec}\,()B, \mathtt{Rec}\,()B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \zeta'} \begin{pmatrix}\zeta \subseteq \{\mathrm{pos}\} \\ \vee\ \zeta' = \{\,\}\end{pmatrix}$$

$$\frac{B,B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,1,\mathrm{pos}}\triangleright \zeta \qquad B,B' \mathop{\rule{6em}{0.4pt}}\limits^{\mathit{ties}}_{E,<:\mathtt{Top},i+1,v}\triangleright \zeta'}{\mathtt{Rec}\,()B, \mathtt{Rec}\,()B' \mathop{\rule{3em}{0.4pt}}\limits^{\mathit{ties}}_{E,i,v}\triangleright \{\mathrm{pos},\mathrm{neg}\}} \begin{pmatrix}\zeta \not\subseteq \{\mathrm{pos}\} \\ \wedge\ \zeta' \neq \{\,\}\end{pmatrix}$$

G.3. Typing algorithm with argument synthesis

$$\alpha \xrightarrow[m]{\mathit{contracts}} \triangleright \mathrm{ok} \qquad \mathtt{All}\,(q <: A)B \xrightarrow[m]{\mathit{contracts}} \triangleright \mathrm{ok}$$

$$\alpha, B \xrightarrow[E,i,v]{\mathit{ties}} \triangleright \{\} \qquad A, \beta \xrightarrow[E,i,v]{\mathit{ties}} \triangleright \{\}\ (A \neq \gamma) \qquad \dfrac{A', A \xrightarrow[E,i,\neg v]{\mathit{ties}} \triangleright \zeta \qquad B, B' \xrightarrow[E, <: A', i+1, v]{\mathit{ties}} \triangleright \zeta'}{\mathtt{All}\,(q <: A)B, \mathtt{All}\,(q <: A')B' \xrightarrow[E,i,v]{\mathit{ties}} \triangleright \zeta \cup \zeta'}$$

$$\dfrac{B \xrightarrow[i,j+1]{\mathit{lift}} \triangleright B'}{\mathtt{Rec}\,()B \xrightarrow[i,j]{\mathit{lift}} \triangleright \mathtt{Rec}\,()B'} \qquad \dfrac{B \xrightarrow[i+1,C]{\mathit{replace}} \triangleright B'}{\mathtt{Rec}\,()B \xrightarrow[i,C]{\mathit{replace}} \triangleright \mathtt{Rec}\,()B'} \qquad \dfrac{B \xrightarrow[\alpha, r+1, \sigma\Uparrow, i+1]{\mathit{occurCheck}} \triangleright \sigma'}{\mathtt{Rec}\,()B \xrightarrow[\alpha, r, \sigma, i]{\mathit{occurCheck}} \triangleright \sigma'\Downarrow}$$

$$\dfrac{T \xrightarrow[E, <: \mathtt{Top}]{\mathit{type}} \triangleright B \qquad B \xrightarrow{\mathit{contracts}} \triangleright \mathrm{ok}}{\mathtt{Rec}\,()T \xrightarrow[E]{\mathit{type}} \triangleright \mathtt{Rec}\,()B} \qquad \dfrac{B, C \xrightarrow[E, <: \mathtt{Top}]{\mathit{ties}} \triangleright \zeta \qquad B, C \xrightarrow[E, <: \mathtt{Top}, \sigma\Uparrow]{\mathit{sub}} \triangleright \sigma'}{\mathtt{Rec}\,()B, \mathtt{Rec}\,()C \xrightarrow[E, \sigma]{\mathit{sub}} \triangleright \sigma'\Downarrow}\ (\zeta \subseteq \{\mathrm{pos}\})$$

$$\dfrac{B, C \xrightarrow[E, <: \mathtt{Top}]{\mathit{ties}} \triangleright \zeta \qquad B, C \xrightarrow[E, <: \mathtt{Top}, \sigma\Uparrow]{\mathit{sub}} \triangleright \sigma' \qquad C, B \xrightarrow[E, <: \mathtt{Top}, \sigma']{\mathit{sub}} \triangleright \sigma''}{\mathtt{Rec}\,()B, \mathtt{Rec}\,()C \xrightarrow[E]{\mathit{sub}} \triangleright \sigma''\Downarrow}\ (\zeta \not\subseteq \{\mathrm{pos}\})$$

$$\dfrac{T \xrightarrow[E]{\mathit{type}} \triangleright \mathtt{Rec}\,()B \qquad a \xrightarrow[E, \sigma]{\mathit{term}} \triangleright A, \sigma' \qquad B \xrightarrow[1, \mathtt{Rec}\,()B]{\mathit{replace}} \triangleright B' \qquad A, B' \xrightarrow[E, \sigma']{\mathit{sub}} \triangleright \sigma''}{\mathtt{fold}\,(:T)\ (a) \xrightarrow[E, \sigma]{\mathit{term}} \triangleright \mathtt{Rec}\,()B, \sigma''}$$

$$\dfrac{a \xrightarrow[E, \sigma]{\mathit{term}} \triangleright A, \sigma' \qquad A \xrightarrow[E, \sigma']{\mathit{exposeRec}} \triangleright \mathtt{Rec}\,()B, \sigma'' \qquad B \xrightarrow[1, \mathtt{Rec}\,()B]{\mathit{replace}} \triangleright B'}{\mathtt{unfold}\,(a) \xrightarrow[E, \sigma]{\mathit{term}} \triangleright B', \sigma''}$$

$$\dfrac{S \xrightarrow[E]{\mathit{type}} \triangleright A \qquad a \xrightarrow[E, : A, \sigma\Uparrow]{\mathit{term}} \triangleright B, \sigma' \qquad A \xrightarrow[1]{\mathit{lift}} \triangleright A' \qquad B, A' \xrightarrow[E, : A, \sigma']{\mathit{sub}} \triangleright \sigma''}{\mathtt{rec}\,(:S)\ (a) \xrightarrow[E, \sigma]{\mathit{term}} \triangleright A, \sigma''\Downarrow}$$

$$\boxed{type : Type \xrightarrow[env : Env,\ subst : Subst]{exposeRec} \triangleright outType : Type,\ outSubst : Subst}$$

$$\frac{E \xrightarrow[n]{typeIde} \triangleright A \qquad A \xrightarrow[E, \sigma]{exposeRec} \triangleright A', \sigma'}{n \xrightarrow[E, \sigma]{exposeRec} \triangleright A', \sigma'} \qquad \texttt{Rec()}B \xrightarrow[E, \sigma]{exposeRec} \triangleright \texttt{Rec()}B, \sigma$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright A \qquad A \xrightarrow[E, \sigma]{exposeRec} \triangleright A', \sigma'}{\alpha \xrightarrow[E, \sigma]{exposeRec} \triangleright A', \sigma'}$$

$$\frac{\sigma \xrightarrow[\alpha]{retrieve} \triangleright r}{\alpha \xrightarrow[E, \sigma]{exposeRec} \triangleright (\texttt{Rec()}\alpha'), (\alpha \leftarrow (\texttt{Rec()}\alpha'), \alpha'_r, (\sigma \setminus \alpha))} \ (\alpha' \text{ new in } \sigma)$$

Acknowledgments

I would like to thank Mario Coppo, Mariangiola Dezani, and Georges Gonthier for pointing out mistakes in early versions of this paper. I would also like to thank Martín Abadi for discussions on pattern instantiation, and Florian Matthes for discovering a problem with variables that are free in actions. Finally, Alain Knaff designed and implemented certain generalizations of the syntax extensions mechanism described here, and demonstrated that the syntax extension code can be reused for a different language (Prolog).

References

[Amadio, Cardelli 1991] R.M. Amadio and L. Cardelli. **Subtyping recursive types**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*.

[Bove, Arbilla 1992] A. Bove and L. Arbilla. **A confluent calculus of macro expansion and evaluation**. *Proc. 1992 ACM Conference on Lisp and Functional Programming*.

[Cardelli FTP] L. Cardelli, **F-sub**. Anonymous FTP from gatekeeper.pa.dec.com.

[Cardelli, *et al.* 1991] L. Cardelli, J.C. Mitchell, S. Martini, and A. Scedrov. **An extension of system F with subtyping**. *Proc. Theoretical Aspects of Computer Software*. Sendai, Japan. Lecture Notes in Computer Science 526. Springer-Verlag.

[Cardelli, Wegner 1985] L. Cardelli and P. Wegner, **On understanding types, data abstraction and polymorphism**. Computing Surveys **17**(4), 471-522.

[Curien, Ghelli 1991] P.-L. Curien and G. Ghelli. **Subtyping + extensionality: confluence of βη-reductions in F**$_{\leq}$. *Proc. Theoretical Aspects of Computer Software*. Sendai, Japan. Lecture Notes in Computer Science 526. Springer-Verlag.

[Griffin 1988] T.G. Griffin. **Notational definition - a formal account**. *Proc. 3rd Annual IEEE Symposium on Logic in Computer Science*.

[Kalsow, Muller FTP] B. Kalsow and E. Muller, **Modula-3**. Anonymous FTP from gatekeeper.pa.dec.com.

[Kohlbecker, *et al.* 1986] E. Kohlbecker, D.P. Friedman, M. Felleisen, and B. Duba. **Hygienic macro expansion**. *Proc. 1986 ACM Conference on Lisp and Functional Programming*.

[Mauny, Rauglaudre 1992] M. Mauny and D.d. Rauglaudre. **Parsers in ML**. *Proc. 1992 ACM Conference on Lisp and Functional Programming*.

[Miller (to appear)] D. Miller, **Unification under a mixed prefix**. Journal of Symbolic Computation.

[Nelson 1991] G. Nelson, ed. **Systems Programming with Modula-3**. Prentice Hall.

[Pfenning 1989] F. Pfenning. **Elf: a language for logic definition and verified metaprogramming**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*.

[Pierce 1992] B.C. Pierce. **Bounded quantification is undecidable**. *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*.

[Plotkin 1981] G.D. Plotkin. **A structural approach to operational semantics**. Internal Report DAIMI FN-19. Computer Science Department, Aarhus University.

[Pollack 1990] R. Pollack. **Implicit syntax (draft)**. *Proc. First Workshop on Logical Frameworks*.