HOT-Compilation: Phase Splitting

*TA: Akiva Leffert
aleffert@andrew.cmu.edu

Out: Monday, November 13, 2006
Due: Tuesday, November 28, 2006 (Before midnight)

1 Introduction

Many statically typechecked languages like ML enjoy a property referred to as the "phase distinction", which says roughly that the interpretation of a program can be divided into two distinct phases: the static, "compile time" phase, and the dynamic, "run time" phase. The computation of the type of a program cannot depend upon evaluation of the program — one may typecheck a program without executing it.

The phase distinction may seem surprising at first, since ML entangles expressions and types by grouping them together in modules and signatures. A process called phase splitting unravels this coupling, splitting each module and signature into its static components and its dynamic components.

In this assignment, you will implement a phase splitter for a full-featured post-elaboration Standard ML internal language. Your phase splitter will compile away the entire module calculus, including module variables, functors, and sealing, into more primitive notions of binding and abstraction. Each module will become a single constructor and a single expression, and each signature will become a single kind and a single constructor.

2 Overview

Your task is to create a file `convert/phasesplit.sml` defining a structure `PhaseSplit` matching the signature `PHASE_SPLIT` which is found in the file `convert/phasesplit-sig.sml`, reproduced here for your convenience.

```
signature PHASE_SPLIT =
  sig
     exception Split of string
     exception Error of string

     val split_module :  IL1.module -> IL2.module
     val split_sbnds :  IL1.sbnd list
                    -> IL2.con * (IL2.expvar * IL2.exp) list
     val split_exp : IL1.exp -> IL2.exp
     val split_con : IL1.con -> IL2.con
     val split_kind : IL1.kind -> IL2.kind
     val split_signat : IL1.signat -> IL2.signat
     val split_sdecs : IL1.sdec list
                    -> IL2.convar * IL2.kind * IL2.con list
  end
```

*Originally prepared by William Lovas (Fall 2005)

You should implement these phase splitting transformations by following the inference rules in Section 4. Signal any phase splitting errors by raising `Split` with an appropriate error message; you may use the `Error` exception for internal consistency errors such as violated invariants or the occurrence of "impossible" conditions.

The files `il1/il1.sml` and `il2/il2.sml` define the `IL1` and `IL2` structures, which contain all of the datatypes your phase splitter will manipulate. These structures also contain several utility functions for manipulating `IL1` and `IL2` terms that you may find useful, including substitution; you will find their signatures in `il1/il1-sig.sml` and `il2/il2-sig.sml`, respectively.

You may use the test harness interface in the `PhaseSplitTop` structure (see `phasesplittop-sig.sml`) to experiment with your implementation. Several interesting examples are included in the `il1/il1examples.sml` file. A few simple examples to get you started are shown in Section 5.

Submit your code via AFS by copying `phasesplit.sml` to the directory

```
/afs/andrew/course/15/501-819/submit/<your andrew id>/phasesplit/
```

Note: Your submission will be graded automatically; if your submission that fails to compile under SML/NJ 110.59 using this assignment's base distribution, we won't be able to grade it.

3 Syntax

The syntax for IL1 appears in Subsection 3.1. The grammars described correspond quite closely to the datatypes in `il1.sml`, except in their exclusion of existential variables. For this assignment, you may assume that elaboration has removed all evars from the program.

IL1 is similar to the internal language from Project 2 with several extensions, many of which were discussed in class. Some notable additions include:

- mutually recursive functions, $\mathbf{fix}\ (f_i\ (x_i{:}con_i) : con'_i.exp_i)_{i=1}^{n}\ \mathbf{end}$. All of $f_1, \ldots, f_n$ are bound in all of the exp_i, and each x_i is bound in the corresponding exp_i. The whole bundle has a product-of-functions type.

- labelled sums, $+[lab_1{:}con_1, \ldots, lab_n{:}con_n]$. Used as part of the underlying representation of datatypes. As with labelled records, the field order is significant. An expression exp_i of type con_i may be injected into the above type with $\mathbf{inj}_{lab_i}^{+[lab_1:con_1,\ldots,lab_n:con_n],lab_1,\ldots,lab_n}\ exp_i$. An expression e of the above type may be deconstructed using a case construct, $\mathbf{case}^{con}\ exp\ \mathbf{of}\ lab_1 \mapsto exp_1, \ldots, lab_n \mapsto exp_n$, where each of the exp_i has type $con_i \rightarrow con$.

- functors, $\lambda s{:}sig.mod$, and functor signatures $\Pi s{:}sig.sig'$. These are essentially as discussed in class.

- recursive type bundles, $\mu\alpha{:}knd.con$. These are also used in the elaboration of datatypes. The kind knd is restricted to the form $\mathbf{T}^n$. An expression exp_i having type $\pi_i([\mu\alpha{:}knd.con/\alpha]con)$ may be coerced to type $\pi_i(\mu\alpha{:}knd.con)$ by writing $\mathbf{roll}^{\pi_i(\mu\alpha:knd.con)}\ exp_i$. An expression exp'_i of type $\pi_i(\mu\alpha{:}knd.con)$ may be coerced to type $\pi_i([\mu\alpha{:}knd.con/\alpha]con)$ by writing $\mathbf{unroll}\ \mathrm{exp}'_i$.

- functors, $\lambda s{:}sig.mod$, and functor signatures $\Pi s{:}sig.sig'$. These are essentially as discussed in class.

All of the above prose descriptions are formalized in the rules found in Subsection 3.1. Also note that product kinds are now written with a new syntax, $\times[knd_1, \ldots, knd_n]$.

IL2's syntax is found in Subsection 3.2. This language is mostly a subset of IL1, with the following differences:

- Modules and signatures are entirely degenerate: there are no notions of computation over modules, not even projection of their components. What vestiges of modules remain are there simply to represent complete programs.

- Products of kinds $\times[knd_1, \ldots, knd_n]$ and products of constructors $\langle con_1, \ldots, con_n \rangle$ are replaced by the unit kind $\mathbf{1}$ and dependent pair kinds $\Sigma\alpha{:}\kappa_1.\kappa_2$ along with the unit constructor $\star$ and constructor pairs $\langle c_1, c_2 \rangle$. Dependent pair kinds are necessary to express the dependencies from IL1's module language. As usual, we write non-dependent pair kinds as $\kappa_1 \times \kappa_2$. We recover n-ary (non-dependent) product kinds $\times[\kappa_1, \ldots, \kappa_n]$ and n-ary products of constructors $\langle c_1, \ldots, c_n \rangle$ as derived forms by treating them like lists terminated by unit. We write $\pi_1 c$ and $\pi_2 c$ for the ordinary binary first and second projections and $\pi^i c$ for the n-ary analogue.

- Labelled product types $\times[lab_1{:}con_1, \ldots, lab_n{:}con_n]$ and labelled sum types $+[lab_1{:}con_1, \ldots, lab_n{:}con_n]$ are replaced by their unlabelled counterparts, $\times[c_1, \ldots, c_n]$ and $+[c_1, \ldots, c_n]$. The corresponding expression forms also lose their labels, referring to components by (1-indexed) position instead.

3.1 IL1 Syntax

3.1.1 IL1 Kinds

knd	$::=$	$\mathbf{T}$	kind of types
	$\mid$	$\Pi\alpha{:}knd_1.knd_2$	dependent function kinds
	$\mid$	$\times[knd_1, \ldots, knd_n]$	product kinds
	$\mid$	$\mathcal{S}(con)$	singleton kinds

3.1.2 IL1 Constructors

con	$::=$	α	constructor variables
	$\mid$	$\lambda\alpha{:}knd.con$	constructor functions
	$\mid$	$con_1 con_2$	constructor application
	$\mid$	$\langle con_1, \ldots, con_n\rangle$	constructor tuples
	$\mid$	$\pi_i con$	constructor tuple projection
	$\mid$	$\mu\alpha{:}knd.con$	recursive constructors
	$\mid$	${mod_v}^{lab_1,\ldots,lab_n}.lab$	module projections
	$\mid$	$\mathsf{int} \mid \mathsf{char} \mid \mathsf{string}$	built-in base types
	$\mid$	$con_1 \rightarrow con_2$	functions
	$\mid$	$\mathsf{ref}\ con$	references
	$\mid$	tagged	extensible tagged unions
	$\mid$	$\mathsf{tag}\ con$	tags
	$\mid$	$\times[lab_1{:}con_1, \ldots, lab_n{:}con_n]$	labelled products
	$\mid$	$+[lab_1{:}con_1, \ldots, lab_n{:}con_n]$	labelled sums
	$\mid$	$\forall\alpha{:}knd.con$	polymorphic types

3.1.3 IL1 Expressions

exp	$::=$	x	expression variables
	$\mid$	$\overline{n} \mid$ '*char*' $\mid$ "*string*"	integer, character, and string literals
	$\mid$	$\textbf{fix}\ fbnd_1, \ldots, fbnd_n\ \textbf{end}$	mutually recursive function bindings
	$\mid$	$unop\ exp$	built-in unary operations
	$\mid$	$exp_1\ binop\ exp_2$	built-in binary operations
	$\mid$	$exp_1 exp_2$	function applications
	$\mid$	$\textbf{handle}\ exp_1\ \textbf{with}\ exp_2$	exception handlers
	$\mid$	$\textbf{raise}^{con}\ exp$	exception raising
	$\mid$	$\textbf{ref}\ exp$	reference cell allocation
	$\mid$	$\textbf{get}\ exp$	reference cell dereference
	$\mid$	$\textbf{set}(exp_1, exp_2)$	reference cell update
	$\mid$	$\textbf{roll}^{con}\ exp$	coercion into recursive type
	$\mid$	$\textbf{unroll}\ exp$	coercion out of recursive type
	$\mid$	$(lab_1{=}exp_1, \ldots, lab_n{=}exp_n)$	labelled products
	$\mid$	$\pi_{lab}^{lab_1,\ldots,lab_n}\ exp$	product projections
	$\mid$	$\textbf{inj}_{lab}^{con,lab_1,\ldots,lab_n} exp$	sum injection
	$\mid$	$\textbf{case}^{con}\ exp\ \textbf{of}\ lab_1{\mapsto}exp_1, \ldots, lab_n{\mapsto}exp_n$	sum analysis
	$\mid$	$\textbf{tag}(exp_1, exp_2)$	injection into type tagged
	$\mid$	$\textbf{newtag}[con]$	extension of type tagged
	$\mid$	$\textbf{iftagof}\ exp_1\ \textbf{is}\ exp_2\ \textbf{then}\ exp_3\ \textbf{else}\ exp_4$	tag analysis
	$\mid$	$\Lambda\alpha{:}knd.exp$	polymorphic abstraction
	$\mid$	$exp[con]$	polymorphic application
	$\mid$	$\textbf{let}\ x = exp_1\ \textbf{in}\ exp_2$	expression let-binding
	$\mid$	$\textbf{let}\ s = mod\ \textbf{in}\ exp$	module let-binding
	$\mid$	${mod_v}^{lab_1,\ldots,lab_n}.lab$	module projection

3.1.4 IL1 Function Bindings

$fbnd$	$::=$	$f\ (x{:}con) : con'.exp$	function bindings

3.1.5 IL1 Modules

mod	$::=$	s	module variables
	$\mid$	$[sbnd]$	structures
	$\mid$	$\lambda s{:}sig.mod$	functors
	$\mid$	$mod\ mod'_v$	functor application
	$\mid$	${mod_v}^{lab_1,\ldots,lab_n}.lab$	module projection
	$\mid$	$mod :> \sigma$	sealed modules
	$\mid$	$\textbf{let}\ s = mod\ \textbf{in}\ (mod' : \sigma')$	module let-binding

3.1.6 IL1 Bindings

$sbnds$	$::=$	$\cdot \mid sbnd, sbnds$	structure binding list
$sbnd$	$::=$	$lab \triangleright bnd$	structure field bindings
bnd	$::=$	$x = exp$	expression variable bindings
	$\mid$	$\alpha = con$	constructor variable bindings
	$\mid$	$s = mod$	module variable bindings

3.1.7 IL1 Signatures

sig	$::=$	$[sbnds]$	structure signatures
	$\mid$	$\Pi s{:}sig_1.sig_2$	functor signatures

3.1.8 IL1 Declarations

$sdecs$	$::=$	$\cdot \mid sdec, sdecs$	structure declaration list
$sdec$	$::=$	$lab \triangleright dec$	structure field declaration
dec	$::=$	$x{:}con$	expression variable declaration
	$\mid$	$\alpha{:}knd$	constructor variable declaration
	$\mid$	$s{:}sig$	module variable declaration

3.1.9 IL1 Typing Contexts

Γ	$::=$	$\cdot$	empty typing context
	$\mid$	$\Gamma, x{:}con$	expression variable declaration
	$\mid$	$\Gamma, \alpha{:}knd$	constructor variable declaration
	$\mid$	$\Gamma, s{:}sig$	module variable declaration

3.1.10 IL1 Derived Forms

knd^n	$\stackrel{\text{def}}{=}$	$\times[knd, \ldots (n\text{times}) \ldots, knd]$	repeated products
$knd_1 \to knd_2$	$\stackrel{\text{def}}{=}$	$\Pi\_{:}knd_1.knd_2$	non-dependent arrow kinds
unit	$\stackrel{\text{def}}{=}$	$\times[\cdot]$	unit type

3.2 IL2 Syntax

3.2.1 IL2 Kinds

κ	$::=$	$\mathbf{T}$	kind of types
	$\mid$	$\Pi\alpha{:}\kappa_1.\kappa_2$	dependent function kinds
	$\mid$	$\mathbf{1}$	unit kinds
	$\mid$	$\Sigma\alpha{:}\kappa_2.\kappa_2$	dependent pair kinds
	$\mid$	$\mathcal{S}(c)$	singleton kinds

3.2.2 IL2 Constructors

c	$::=$	α	constructor variables
	$\mid$	$\lambda\alpha{:}\kappa.c$	constructor functions
	$\mid$	$c_1 c_2$	constructor application
	$\mid$	$\star$	unit constructor
	$\mid$	$\langle c_1, c_2\rangle$	constructor tuples
	$\mid$	$\pi_1 c$	constructor pair first projection
	$\mid$	$\pi_2 c$	constructor pair second projection
	$\mid$	$\mu\alpha{:}\kappa.c$	recursive constructors
	$\mid$	$\mathsf{int} \mid \mathsf{char} \mid \mathsf{string}$	built-in base types
	$\mid$	$c_1 \rightarrow c_2$	functions
	$\mid$	$\mathsf{ref}\ c$	references
	$\mid$	tagged	extensible tagged unions
	$\mid$	$\mathsf{tag}\ c$	tags
	$\mid$	$\times[c_1, \ldots, c_n]$	labelled products
	$\mid$	$+[c_1, \ldots, c_n]$	labelled sums
	$\mid$	$\forall\alpha{:}\kappa.c$	polymorphic types

3.2.3 IL2 Expressions

e	$::=$	x	expression variables
	$\mid$	$\overline{n} \mid \text{'}\mathit{char}\text{'} \mid \text{"}\mathit{string}\text{"}$	integer, character, and string literals
	$\mid$	$\lambda x{:}con.exp$	non-recursive anonymous functions
	$\mid$	$\textbf{fix}\ fbnd_1, \ldots, fbnd_n\ \textbf{end}$	mutually recursive function bindings
	$\mid$	$unop\ e$	built-in unary operations
	$\mid$	$e_1\ binop\ e_2$	built-in binary operations
	$\mid$	$e_1 e_2$	function applications
	$\mid$	$\textbf{handle}\ e_1\ \textbf{with}\ e_2$	exception handlers
	$\mid$	$\textbf{raise}^c\ e$	exception raising
	$\mid$	$\textbf{ref}\ e$	reference cell allocation
	$\mid$	$\textbf{get}\ e$	reference cell dereference
	$\mid$	$\textbf{set}(e_1, e_2)$	reference cell update
	$\mid$	$\textbf{roll}^c\ e$	coercion into recursive type
	$\mid$	$\textbf{unroll}\ e$	coercion out of recursive type
	$\mid$	$(e_1, \ldots, e_n)$	unlabelled products
	$\mid$	$\pi_i\ e$	product projections
	$\mid$	$\textbf{inj}_i^c e$	sum injection
	$\mid$	$\textbf{case}^c\ e\ \textbf{of}\ e_1, \ldots, e_n$	sum analysis
	$\mid$	$\textbf{tag}(e_1, e_2)$	injection into type `tagged`
	$\mid$	$\textbf{newtag}[c]$	extension of type `tagged`
	$\mid$	$\textbf{iftagof}\ e_1\ \textbf{is}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4$	tag analysis
	$\mid$	$\Lambda\alpha{:}\kappa.e$	polymorphic abstraction
	$\mid$	$e[c]$	polymorphic application
	$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	expression let-binding

3.2.4 IL2 Function Bindings

fb	$::=$	$f\ (x{:}c) : c'.e$	function bindings

3.2.5 IL2 Vestigal Module System

m	$::=$	$[c, e]$	phase split structures
σ	$::=$	$[\alpha{:}\kappa.c]$	phase split signatures

3.2.6 IL2 Derived Forms

κs	$::=$	$\cdot \mid \kappa, \kappa s$	kind lists
cs	$::=$	$\cdot \mid c, cs$	constructor lists
$ebnds$	$::=$	$\cdot \mid x = e, ebnds$	expression binding lists
$\kappa_1 \to \kappa_2$	$\stackrel{\text{def}}{=}$	$\Pi\_{:}\kappa_1.\kappa_2$	non-dependent function kinds
$\times[\kappa_1]\kappa_2$	$\stackrel{\text{def}}{=}$	$\Sigma\_{:}\kappa_1.\kappa_2$	non-dependent product kinds
$\times[\cdot]$	$\stackrel{\text{def}}{=}$	$\mathbf{1}$	0-ary product kind
$\times[\kappa, \kappa s]$	$\stackrel{\text{def}}{=}$	$\kappa \times \times[\kappa s]$	non-empty n-ary product kinds
$\times[\cdot]$	$\stackrel{\text{def}}{=}$	$\star$	0-ary product of constructors
$\times[c, cs]$	$\stackrel{\text{def}}{=}$	$\times[c, \times[cs]]$	non-empty n-ary product of constructors
$\pi^1 c$	$\stackrel{\text{def}}{=}$	$\pi_1 c$	first projection from n-ary product
$\pi^i c$	$\stackrel{\text{def}}{=}$	$\pi^{i-1}(\pi_2 c)$	ith projection from n-ary product

4 Phase Splitting

Phase splitting is presented here as a syntax-derivation-directed translation as in class. One minor difference from what was presented in class is that we make use here of n-ary product types and tuple expressions rather than just pairs. (Using pairs at the expression level would actually make the code generated by our compiler less efficient, since it would have to perform linear scans through phase-split modules to access their components.) Accordingly, the *sdecs*-splitting judgment returns not just one type with a free variable, but rather a list of types with a free variable. Similarly, the *sbnds*-splitting judgment returns a list of variable-expression bindings instead of building up one large expression using **let**-binding. The *sig*- and *mod*-splitting judgments account for these differences by packaging the result up in the final form we expect.

4.1 Signatures

$\boxed{P(sig) = \sigma}$

$$\frac{P(sdecs) = \alpha{:}\kappa.c_1, \ldots, c_n}{P([sdecs]) = [\alpha{:}\kappa. \times [c_1, \ldots, c_n]]} \tag{1}$$

Rule (1): As mentioned above, this judgment packages the list of types returned by the *sdecs*-splitting judgment into a product.

$$\frac{P(sig_1) = [\alpha_1{:}\kappa_1.c_1] \quad P(sig_2) = [\alpha_2{:}\kappa_2.c_2]}{P(\Pi s{:}sig_1.sig_2) = [\beta{:}(\Pi\alpha_s{:}\kappa_1.\kappa_2).\forall\alpha_s{:}\kappa_1.[\alpha_s/\alpha_1]c_1 \to [\beta\alpha_s/\alpha_2]c_2]} \tag{2}$$

$\boxed{P(sdecs) = \alpha{:}\kappa.cs}$

$$\frac{}{P(\cdot) = \alpha{:}\mathbf{1}.\cdot} \tag{3}$$

$$\frac{P(con) = c \quad P(sdecs) = \beta{:}\kappa.cs}{P(lab \triangleright x{:}con, sdecs) = \alpha{:}(\mathbf{1} \times \kappa).c, [\pi_2\alpha/\beta]cs} \tag{4}$$

$$\frac{P(knd) = \kappa' \quad P(sdecs) = \gamma{:}\kappa.cs}{P(lab \triangleright \alpha{:}knd, sdecs) = \beta{:}(\Sigma\alpha{:}\kappa'.\kappa).\mathsf{unit}, [\pi_1\beta, \pi_2\beta/\alpha, \gamma]cs} \tag{5}$$

$$\frac{P(sig) = [\delta{:}\kappa'.c] \quad P(sdecs) = \gamma{:}\kappa.cs}{P(lab \triangleright s{:}sig, sdecs) = \beta{:}(\Sigma\alpha_s{:}\kappa'.\kappa).[\pi_1\beta/\delta]c, [\pi_1\beta, \pi_2\beta/\alpha_s, \gamma]cs} \tag{6}$$

4.2 Modules

$\boxed{P(mod) = m}$

$$\frac{}{P(s) = [\alpha_s, x_s]} \tag{7}$$

Rule (7): For each module variable s we create a constructor variable α_s and an expression variable x_s. In your implementation you should maintain a mapping from module variables to constructor variables and expression variables, creating α_s and x_s fresh the first time you need them.

$$\frac{P(sbnds) = c; x_1{=}e_1, \ldots, x_n{=}e_n}{P([sbnds]) = [c, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ \ldots \mathbf{let}\ x_n = e_n\ \mathbf{in}\ (x_1, \ldots, x_n)]} \tag{8}$$

Rule (8): Here we **let**-bind the variable-expression bindings returned by the *sbnds*-splitting judgment.

$$\frac{P(sig) = [\alpha{:}\kappa.c] \quad P(mod) = [c', e]}{P(\lambda s{:}sig.mod) = [\lambda\alpha_s{:}\kappa.c', \Lambda\alpha_s{:}\kappa.\lambda x_s{:}[\alpha_s/\alpha]c.e]} \tag{9}$$

$$\frac{P(mod) = [c, e] \quad P(mod_v') = [c', e']}{P(mod\ mod_v') = [cc', e[c']e']} \tag{10}$$

$$\frac{P(mod_v) = [c, e] \quad lab = lab_i}{P(mod_v{}^{lab_1,\ldots,lab_n}.lab) = [\pi^i c, \pi_i\ e]} \tag{11}$$

$$\frac{P(mod) = m}{P(mod :> sig) = m} \tag{12}$$

$$\frac{P(mod) = [c, e] \quad P(mod') = [c', e']}{P(\textbf{let}\ s = mod\ \textbf{in}\ (mod' : sig')) = [[c/\alpha_s]c', \textbf{let}\ x_s = e\ \textbf{in}\ [c/\alpha_s]e']} \tag{13}$$

$\boxed{P(sbnds) = c; ebnds}$

$$\frac{}{P(\cdot) = \star; \cdot} \tag{14}$$

$$\frac{P(exp) = e \quad P(sbnds) = c; ebnds}{P(lab \triangleright x = exp, sbnds) = \langle \star, c\rangle; x = e, ebnds} \tag{15}$$

$$\frac{P(con) = c' \quad P(sbnds) = c; ebnds}{P(lab \triangleright \alpha = con, sbnds) = \langle c', [c'/\alpha]c\rangle; \_ = (), [c'/\alpha]ebnds} \tag{16}$$

Rule (16): A con has no dynamic part so it is okay to just make up a fresh variable for the *ebnd* part.

$$\frac{P(mod) = [c', e'] \quad P(sbnds) = c; ebnds}{P(lab \triangleright s = mod, sbnds) = \langle c', [c'/\alpha_s]c\rangle; x_s = e', [c'/\alpha_s]ebnds} \tag{17}$$

4.3 Kinds

$\boxed{P(knd) = \kappa}$

$$\frac{}{P(\mathbf{T}) = \mathbf{T}} \tag{18}$$

$$\frac{P(knd_1) = \kappa_1 \quad P(knd_2) = \kappa_2}{P(\Pi\alpha{:}knd_1.knd_2) = \Pi\alpha{:}\kappa_1.\kappa_2} \tag{19}$$

$$\frac{P(knd_i) = \kappa_i \quad \text{forall}_i\ 1 \le i \le n}{P(\times[knd_1, \ldots, knd_n]) = \times[\kappa_1, \ldots, \kappa_n]} \tag{20}$$

Rule (20): We make use here of the n-ary product kind derived form.

$$\frac{P(con) = c}{P(\mathcal{S}(con)) = \mathcal{S}(c)} \tag{21}$$

4.4 Constructors

$\boxed{P(con) = c}$

$$\frac{}{P(\alpha) = \alpha} \tag{22}$$

$$\frac{P(knd) = \kappa \quad P(con) = c}{P(\lambda\alpha{:}knd.con) = \lambda\alpha{:}\kappa.c} \tag{23}$$

$$\frac{P(con_1) = c_1 \quad P(con_2) = c_2}{P(con_1 con_2) = c_1 c_2} \tag{24}$$

$$\frac{P(con_i) = c_i \quad \text{forall}_i\ 1 \le i \le n}{P(\langle con_1, \ldots, con_n\rangle) = \langle c_1, \ldots, c_n\rangle} \tag{25}$$

$$\frac{P(con) = c}{P(\pi_i con) = \pi^i c} \tag{26}$$

$$\frac{P(knd) = \kappa \quad P(con) = c}{P(\mu\alpha{:}knd.con) = \mu\alpha{:}\kappa.c} \tag{27}$$

$$\frac{P(mod_v) = [c, e] \quad lab = lab_i}{P({mod_v}^{lab_1,\ldots,lab_n}.lab_i) = \pi^i c} \tag{28}$$

$$\frac{}{P(\mathsf{int}) = \mathsf{int}} \tag{29}$$

$$\frac{}{P(\mathsf{char}) = \mathsf{char}} \tag{30}$$

$$\frac{}{P(\mathsf{string}) = \mathsf{string}} \tag{31}$$

$$\frac{P(con_1) = c_1 \quad P(con_2) = c_2}{P(con_1 \to con_2) = c_1 \to c_2} \tag{32}$$

$$\frac{P(con) = c}{P(\mathsf{ref}\ con) = \mathsf{ref}\ c} \tag{33}$$

$$\frac{}{P(\mathsf{tagged}) = \mathsf{tagged}} \tag{34}$$

$$\frac{P(con) = c}{P(\mathsf{tag}\ con) = \mathsf{tag}\ c} \tag{35}$$

$$\frac{P(con_i) = c_i \quad \text{forall}_i\ 1 \le i \le n}{P(+[lab_1{:}con_1, \ldots, lab_n{:}con_n]) = +[c_1, \ldots, c_n]} \tag{36}$$

$$\frac{P(con_i) = c_i \quad \text{forall}_i\ 1 \le i \le n}{P(\times[lab_1{:}con_1, \ldots, lab_n{:}con_n]) = \times[c_1, \ldots, c_n]} \tag{37}$$

Rules (36) and (37): Since the fields in labelled sums and products are ordered, we take this opportunity to erase the labels.

$$\frac{P(knd) = \kappa \quad P(con) = c}{P(\forall\alpha{:}knd.con) = \forall\alpha{:}\kappa.c} \tag{38}$$

4.5 Expressions

$\boxed{P(exp) = e}$

$$\frac{}{P(x) = x} \tag{39}$$

$$\frac{}{P(\overline{n}) = \overline{n}} \tag{40}$$

$$\frac{}{P(\text{‘}char\text{’}) = \text{‘}char\text{’}} \tag{41}$$

$$\frac{}{P(\text{“}string\text{”}) = \text{“}string\text{”}} \tag{42}$$

$$\frac{P(con_i) = c_i \quad P(con'_i) = c'_i \quad P(exp_i) = e_i \quad \text{forall}_i\ 1 \le i \le n}{P(\mathbf{fix}\ [f_i\ (x_i{:}con_i) : con'_i.exp_i]_{i=1}^n\ \mathbf{end}) = \mathbf{fix}\ [f_i\ (x_i{:}c_i) : c'_i.e_i]_{i=1}^n\ \mathbf{end}} \tag{43}$$

$$\frac{P(exp_1) = e_1 \quad P(exp_2) = e_2}{P(exp_1\ binop\ exp_2) = e_1\ binop\ e_2} \tag{44}$$

$$\frac{P(exp) = e}{P(unop\ exp) = unop\ e} \tag{45}$$

$$\frac{P(exp_1) = e_1 \quad P(exp_2) = e_2}{P(exp_1 exp_2) = e_1 e_2} \tag{46}$$

$$\frac{P(exp_1) = e_1 \quad P(exp_2) = e_2}{P(\textbf{handle } exp_1 \textbf{ with } exp_2) = \textbf{handle } e_1 \textbf{ with } e_2} \tag{47}$$

$$\frac{P(con) = c \quad P(exp) = e}{P(\textbf{raise}^{con}\ exp) = \textbf{raise}^{c}\ e} \tag{48}$$

$$\frac{P(exp) = e}{P(\textbf{ref } exp) = \textbf{ref } e} \tag{49}$$

$$\frac{P(exp) = e}{P(\textbf{get } exp) = \textbf{get } e} \tag{50}$$

$$\frac{P(exp_1) = e \quad P(exp_2) = e_2}{P(\textbf{set}(exp_1, exp_2)) = \textbf{set}(e_1, e_2)} \tag{51}$$

$$\frac{P(con) = c \quad P(exp) = e}{P(\textbf{roll}^{con}\ exp) = \textbf{roll}^{c}\ e} \tag{52}$$

$$\frac{P(exp) = e}{P(\textbf{unroll } exp) = \textbf{unroll } e} \tag{53}$$

$$\frac{P(exp_i) = e_i \quad \text{forall}_i\ 1 \leq i \leq n}{P((lab_1{=}exp_1, \ldots, lab_n{=}exp_n)) = (e_1, \ldots, e_n)} \tag{54}$$

$$\frac{P(exp) = e \quad lab = lab_i}{P(\pi_{lab}^{lab_1, \ldots, lab_n}\ exp) = \pi_i\ e} \tag{55}$$

$$\frac{lab = lab_i \quad P(con) = c \quad P(exp) = e}{P(\textbf{inj}_{lab}^{con, lab_1, \ldots, lab_n} exp) = \textbf{inj}_i^c e} \tag{56}$$

$$\frac{\begin{array}{c} P(con) = c \qquad P(exp) = e \\ P(exp_i) = e_i \quad \text{forall}_i\ 1 \leq i \leq n \end{array}}{P(\textbf{case}^{con}\ exp \textbf{ of } lab_1{\to}exp_1, \ldots, lab_n{\to}exp_n) = \textbf{case}^c\ e \textbf{ of } e_1, \ldots, e_n} \tag{57}$$

$$\frac{P(exp_1) = e_1 \quad P(exp_2) = e_2}{P(\textbf{tag}(exp_1, exp_2)) = \textbf{tag}(e_1, e_2)} \tag{58}$$

$$\frac{P(con) = c}{P(\textbf{newtag}[con]) = \textbf{newtag}[c]} \tag{59}$$

$$\frac{P(exp_1) = e_1 \quad P(exp_2) = e_2 \quad P(exp_4) = e_4 \quad P(exp_3) = e_3}{P(\textbf{iftagof } exp_1 \textbf{ is } exp_2 \textbf{ then } exp_3 \textbf{ else } exp_4) = \textbf{iftagof } e_1 \textbf{ is } e_2 \textbf{ then } e_3 \textbf{ else } e_4} \tag{60}$$

$$\frac{P(knd) = \kappa \quad P(exp) = e}{P(\Lambda\alpha{:}knd.exp) = \Lambda\alpha{:}\kappa.e} \tag{61}$$

$$\frac{P(exp) = e \quad P(con) = c}{P(exp[con]) = e[c]} \tag{62}$$

$$\frac{P(exp) = cone \quad P(exp') = e'}{P(\textbf{let } x = exp \textbf{ in } exp') = \textbf{let } x = e \textbf{ in } e'} \tag{63}$$

$$\frac{P(exp) = e \quad P(mod) = [c', e']}{P(\textbf{let } s = mod \textbf{ in } exp) = \textbf{let } x_s = e' \textbf{ in } [c'/\alpha_s]e} \tag{64}$$

$$\frac{P(mod_v) = [c, e] \quad lab = lab_i}{P({mod_v}^{lab_1, \ldots, lab_n}.lab_i) = \pi_i\ e} \tag{65}$$

5 Examples

The following interactions with the SML/NJ top-level will give you you some simple examples how the splitter should work. More sample inputs may be found in the `IL1Examples` structure defined in `il1/il1examples.sml`.

5.1 Kinds

```
- PhaseSplitTop.split_kind IL1Examples.kind;
  |- *[S(*[a : int, b : +[1 : *[], 2 : *[]]]), Type, S(char)]
 ~~> Sigma __28:S(*[int, +[*[], *[]]]).
     Sigma __27:Type. Sigma __26:S(char). 1
val it = () : unit
```

5.2 Constructors

```
- PhaseSplitTop.split_con IL1Examples.con;
  |- #3 <int, char, string, +[1 : *[], 2 : *[]]>
 ~~> #1 (#2 (#2 <int, <char, <string, <+[*[], *[]], <>>>>>))
val it = () : unit
```

5.3 Expressions

```
- PhaseSplitTop.split_exp IL1Examples.exp1;
  |- #b [a, b, c] {a = 12, b = 'c', c = "xyzzy"}
 ~~> #2 (12, 'c', "xyzzy")
val it = () : unit
```

5.4 Signatures

```
- PhaseSplitTop.split_signat IL1Examples.signat1;
  |- [t |> t_0 : Type,
      x |> x_2 : t_0,
      u |> u_1 : S(*[1 : t_0, 2 : int]),
      y |> y_3 : u_1]
 ~~> [a'_39 :
      (Sigma t_29:Type.
       Sigma __38:1. Sigma u_31:S(*[t_29, int]). Sigma __35:1. 1).
      *[*[], #1 a'_39, *[], #1 (#2 (#2 a'_39))]]
val it = () : unit
```

5.5 Modules

```
- PhaseSplitTop.split_module IL1Examples.module1;
  |- [t |> t_0 = string,
      x |> x_2 = "hello",
      u |> u_1 = *[1 : t_0, 2 : int],
      y |> y_3 = {1 = x_2, 2 = 12}]
 ~~> [<string, <<>, <*[string, int], <<>, <>>>>>,
      let var_45 = ()
      in
          let x_41 = "hello"
          in
```

```
              let var_44 = ()
              in
                  let y_43 = (x_41, 12)
                  in
                      (var_45, x_41, var_44, y_43)
                  end
              end
          end
      end]
val it = () : unit
```