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.

```sml
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 \texttt{Split} with an appropriate error message: you may use the \texttt{Error} exception for internal consistency errors such as violated invariants or the occurrence of “impossible” conditions.

The files \texttt{i11/i11.sml} and \texttt{i12/i12.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 \texttt{i11/i11-sig.sml} and \texttt{i12/i12-sig.sml}, respectively.

You may use the test harness interface in the \texttt{PhaseSplitTop} structure (see \texttt{phasesplittop-sig.sml}) to experiment with your implementation. Several interesting examples are included in the \texttt{i11/i1iexamples.sml} file. A few simple examples to get you started are shown in Section 5.

Submit your code via AFS by copying \texttt{phasesplit.sml} to the directory

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

\textbf{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 \texttt{i11.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, \texttt{fix \{ f_1 \ldots f_n \ : \ \text{con}_1 \ldots \text{con}_n \} \ \text{where} \ \exp_1 \ldots \exp_n \ \text{end}}. All of \( f_1, \ldots, f_n \) are bound in all of the \( \exp_1 \), and each \( x_i \) is bound in the corresponding \( \exp_i \). The whole bundle has a product-of-functions type.
- labelled sums, \(+[\text{lab}_1;\text{con}_1], \ldots,\text{lab}_n;\text{con}_n]\). Used as part of the underlying representation of datatypes. As with labelled records, the field order is significant. An expression \( \exp_i \) of type \( \text{con}_i \) may be injected into the above type with \texttt{inj}_{\text{lab}_1;\text{con}_1} \ldots \text{lab}_n;\text{con}_n \exp_i \). An expression \( e \) of the above type may be deconstructed using a case construct, \texttt{case}^{\text{con}} \exp \text{ of } \text{lab}_1 \mapsto \exp_1, \ldots, \text{lab}_n \mapsto \exp_n, \text{ where each of } \exp_i \text{ has type } \text{con}_i \rightarrow \text{con}.
- functors, \( \lambda s : \text{sig.mod} \), and functor signatures \( \Pi : \text{sig.sig}' \). These are essentially as discussed in class.
- recursive type bundles, \( \mu \alpha : \text{knd.con} \). These are also used in the elaboration of datatypes. The kind \( \text{knd} \) is restricted to the form \( \text{T}' \). An expression \( \exp \) having type \( \pi_i(\mu \alpha : \text{knd.con}/\alpha|\text{con}) \) may be coerced to type \( \pi_i(\mu \alpha : \text{knd.con} \ldots \text{knd}_n) \) by writing \texttt{roll}(\mu \alpha : \text{knd.con}) \exp. An expression \( \exp_i' \) of type \( \pi_i(\mu \alpha : \text{knd.con}) \) may be coerced to type \( \pi_i((\mu \alpha : \text{knd.con}/\alpha)|\text{con}) \) by writing \texttt{unroll} \exp_i'.
- functors, \( \lambda s : \text{sig.mod} \), and functor signatures \( \Pi : \text{sig.sig}' \). 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, \( \times[\text{knd}_1, \ldots, \text{knd}_n] \).

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 $\times[k_{n_1},\ldots,k_{n_d}]$ and products of constructors $\langle con_1,\ldots,con_n \rangle$ are replaced by the unit kind $1$ and dependent pair kinds $\Sigma \alpha : \kappa_1. \kappa_2$ along with the unit constructor $\ast$ 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 $\pi_1c$ and $\pi_2c$ 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

\[ \text{knd ::= T} \mid \Pi \alpha: \text{knd}_1, \text{knd}_2 \mid \times [\text{knd}_1, \ldots, \text{knd}_n] \mid S(\text{con}) \]

3.1.2 IL1 Constructors

\[ \text{con ::= } \alpha \mid \lambda \alpha: \text{knd}. \text{con} \mid \text{con}_1 \text{con}_2 \mid \langle \text{con}_1, \ldots, \text{con}_n \rangle \mid \pi_i \text{con} \mid \mu \alpha: \text{knd}. \text{con} \mid \text{mod}_v \text{lab}_1, \ldots, \text{lab}_n. \text{lab} \mid \text{int} \mid \text{char} \mid \text{string} \mid \text{con}_1 \rightarrow \text{con}_2 \mid \text{ref con} \mid \text{tagged} \mid \text{tag con} \mid \times [\text{lab}_1: \text{con}_1, \ldots, \text{lab}_n: \text{con}_n] \mid + [\text{lab}_1: \text{con}_1, \ldots, \text{lab}_n: \text{con}_n] \mid \forall \alpha: \text{knd}. \text{con} \]

kind of types
dependent function kinds
product kinds
singleton kinds
constructor variables
constructor functions
constructor application
constructor tuples
constructor tuple projection
recursive constructors
module projections
built-in base types
functions
references
extensible tagged unions
tags
labelled products
labelled sums
polymorphic types
3.1.3 IL1 Expressions

\[
\text{exp ::= } x \mid \pi \mid \text{'char'} \mid \text{"string"} \mid \text{fix } fbnd_1, \ldots, fbnd_n \text{ end} \mid \text{unop } exp \mid \text{exp}_1 \text{ binop } exp_2 \mid \text{exp}_1 \text{ exp}_2 \mid \text{handle } exp_1 \text{ with } exp_2 \mid \text{raise}^\text{con} \text{ exp} \mid \text{ref } exp \mid \text{get } exp \mid \text{set}(exp_1, exp_2) \mid \text{roll}^\text{con} \text{ exp} \mid \text{unroll} \text{ exp} \mid \text{handle} (\text{lab}_1 = \text{exp}_1, \ldots, \text{lab}_n = \text{exp}_n) \mid \pi^\text{lab}_1,\ldots,\text{lab}_n \text{ exp} \mid \text{inj}^\text{lab}_1,\ldots,\text{lab}_n \text{ exp} \mid \text{case}^\text{con} \text{ exp of } \text{lab}_1 \mapsto \text{exp}_1, \ldots, \text{lab}_n \mapsto \text{exp}_n \mid \text{tag}(exp_1, exp_2) \mid \text{newtag}[\text{con}] \mid \text{iftagof } exp_1 \text{ is } exp_2 \text{ then } exp_3 \text{ else } exp_4 \mid \Lambda \alpha: \text{knd. } exp \mid \text{exp[con]} \mid \text{let } x = \text{exp}_1 \text{ in } exp_2 \mid \text{let } s = \text{mod in } exp \mid \text{mod}^\text{v,lab}_1,\ldots,\text{lab}_n \text{.lab}
\]

3.1.4 IL1 Function Bindings

\[
\text{fbnd ::= } f (x: \text{con}) : \text{con'}. \text{exp}
\]

3.1.5 IL1 Modules

\[
\text{mod ::= } s \mid [\text{sbnd}] \mid \lambda s : \text{sig. } \text{mod} \mid \text{mod} \text{ mod}_v \mid \text{mod}^\text{lab}_1,\ldots,\text{lab}_n \text{.lab} \mid \text{mod} : > \sigma \mid \text{let } s = \text{mod in } (\text{mod'} : \sigma')
\]

3.1.6 IL1 Bindings

\[
\text{sbnds ::= } \cdot \mid \text{sbnd, sbnds} \mid \text{lab} > \text{bnd} \mid x = \text{exp} \mid \alpha = \text{con} \mid s = \text{mod}
\]

3.1.7 IL1 Signatures

\[
\text{sig ::= } [\text{sbnds}] \mid \Pi s : \text{sig}_1, \text{sig}_2
\]

expression variables
integer, character, and string literals
mutually recursive function bindings
built-in unary operations
built-in binary operations
function applications
exception handlers
exception raising
reference cell allocation
reference cell dereference
reference cell update
coercion into recursive type
coercion out of recursive type
labelled products
product projections
sum injection
sum analysis
injection into type tagged
extension of type tagged
tag analysis
polymorphic abstraction
polymorphic application
expression let-binding
module let-binding
module projection

function bindings
module variables
structures
functors
functor application
module projection
sealed modules
module let-binding

structure binding list
expression field bindings
expression variable bindings
constructor variable bindings
module variable bindings

structure signatures
functor signatures
3.1.8 IL1 Declarations

\[
\begin{align*}
\text{sdecs} &::= \cdot \mid \text{sdec}, \text{sdecs} \\
\text{sdec} &::= \text{lab} \triangleright \text{dec} \\
\text{dec} &::= x:\text{con} \\
&\quad \mid \alpha:\text{knd} \\
&\quad \mid s:\text{sig}
\end{align*}
\]

structure declaration list
structure field declaration
expression variable declaration
constructor variable declaration
module variable declaration

3.1.9 IL1 Typing Contexts

\[
\Gamma ::= \cdot \\
\quad \mid \Gamma, x:\text{con} \\
\quad \mid \Gamma, \alpha:\text{knd} \\
\quad \mid \Gamma, s:\text{sig}
\]

empty typing context
expression variable declaration
constructor variable declaration
module variable declaration

3.1.10 IL1 Derived Forms

\[
\begin{align*}
k\text{nd}^n &\overset{\text{def}}{=} \times[k\text{nd}, \ldots (n\text{times})\ldots , k\text{nd}] \\
k\text{nd}_1 \rightarrow k\text{nd}_2 &\overset{\text{def}}{=} \Pi_{\kappa} k\text{nd}_1, k\text{nd}_2 \\
\text{unit} &\overset{\text{def}}{=} \times[\cdot]
\end{align*}
\]

repeated products
non-dependent arrow kinds
unit type
3.2 IL2 Syntax

3.2.1 IL2 Kinds

| $\kappa ::= \mathbf{T}$              | kind of types                  |
| $\Pi\alpha:\kappa_1,\kappa_2$       | dependent function kinds       |
| $\mathbf{1}$                        | unit kinds                     |
| $\Sigma\alpha:\kappa_2,\kappa_2$   | dependent pair kinds           |
| $S(c)$                              | singleton kinds                |

3.2.2 IL2 Constructors

| $c ::= \alpha$                      | constructor variables          |
| $\lambda\alpha:\kappa.c$            | constructor functions           |
| $c_1c_2$                            | constructor application         |
| $\ast$                               | unit constructor                |
| $\langle c_1, c_2 \rangle$          | constructor tuples              |
| $\pi_1c$                            | constructor pair first projection|
| $\pi_2c$                            | constructor pair second projection|
| $\mu\alpha:\kappa.c$                | recursive constructors          |
| $\text{int} | \text{char} | \text{string}$                  | built-in base types             |
| $c_1 \rightarrow c_2$               | functions                       |
| $\text{ref}c$                       | references                      |
| $\text{tagged}$                     | extensible tagged unions        |
| $\text{tag}c$                       | tags                            |
| $\times[c_1, \ldots, c_n]$          | labelled products               |
| $+[c_1, \ldots, c_n]$               | labelled sums                   |
| $\forall\alpha:\kappa.c$            | polymorphic types               |
3.2.3 IL2 Expressions

\[
e ::= x \mid \pi \mid \text{char}' \mid \text{"string"} \mid \lambda x\text{.}e \mid \text{fix } f\text{bind}_1, \ldots, f\text{bind}_n \text{ end} \mid \text{unop } e \mid e_1 \binom op e_2 \mid e_1 e_2 \mid \text{handle } e_1 \text{ with } e_2 \mid \text{raise}^e e \mid \text{ref } e \mid \text{get } e \mid \text{set} (e_1, e_2) \mid \text{roll}^e e \mid \text{unroll } e \mid (e_1, \ldots, e_n) \mid \pi_i e \mid \text{case}^e e \text{ of } e_1, \ldots, e_n \mid \text{tag}(e_1, e_2) \mid \text{newtag}[c] \mid \text{iftagof } e_1 \text{ is } e_2 \text{ then } e_3 \text{ else } e_4 \mid \Lambda \alpha: \kappa.e \mid e[c] \mid \text{let } x = e_1 \text{ in } e_2
\]

3.2.4 IL2 Function Bindings

\[
f b ::= f (x : c) : c'.e
\]

3.2.5 IL2 Vestigal Module System

\[
m ::= [c, e] \quad \sigma ::= [\alpha : \kappa . c]
\]

3.2.6 IL2 Derived Forms

\[
\kappa s ::= \cdot | \kappa, \kappa s \quad c s ::= \cdot | c, c s \quad e b n d s ::= \cdot | x = e, e b n d s \\
\kappa_1 \to \kappa_2 \triangleq \Pi \kappa_1, \kappa_2 \quad \times [\kappa, \kappa s] \triangleq \kappa \times \times [\kappa s] \\
\times [\cdot] \triangleq \cdot \quad \times [\cdot] \triangleq * \\
\pi^1 e \triangleq \times [c, \times [c s]] \quad \pi^i c \triangleq \pi^{i-1}(\pi_2 c)
\]

expression variables
integer, character, and string literals
non-recursive anonymous functions
mutually recursive function bindings
built-in unary operations
built-in binary operations
function applications
exception handlers
exception raising
reference cell allocation
reference cell dereference
reference cell update
coercion into recursive type
coercion out of recursive type
unlabelled products
product projections
sum injection
sum analysis
injection into type tagged
extension of type tagged
tag analysis
polymorphic abstraction
polymorphic application
expression let-binding

function bindings

phase split structures

phase split signatures

kind lists
constructor lists
expression binding lists
non-dependent function kinds
non-dependent product kinds
0-ary product kind
non-empty n-ary product kinds
0-ary product of constructors
non-empty n-ary product of constructors
first projection from n-ary product
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 \( \text{sdecs} \)-splitting judgment returns not just one type with a free variable, but rather a list of types with a free variable. Similarly, the \( \text{sbnds} \)-splitting judgment returns a list of variable-expression bindings instead of building up one large expression using \texttt{let}-binding. The \( \text{sig} \)- and \( \text{mod} \)-splitting judgments account for these differences by packaging the result up in the final form we expect.

4.1 Signatures

\[
P(\text{sig}) = \sigma
\]

\[P(\text{sig}_1) = [\alpha_1: \kappa_1. c_1] \quad P(\text{sig}_2) = [\alpha_2: \kappa_2. c_2]
\]

\[
P(\Pi: \text{sig}_1, \text{sig}_2) = [\beta:(\Pi \alpha_1: \kappa_1. c_1 \times \alpha_2: \kappa_2. c_2)]
\]

\[P(\text{sdecs}) = \alpha: \kappa. cs
\]

\[
P(\cdot) = \alpha: 1.
\]

\[P(\text{con}) = e \quad P(\text{sdecs}) = \beta: \kappa. cs
\]

\[
P(\text{lab} \triangleright x: \text{con}, \text{sdecs}) = \alpha: ([1 \times \kappa]. e, \pi_2 \alpha / \beta) \cs
\]

\[P(\text{mod} \triangleright \alpha: \text{knd}, \text{sdecs}) = \beta: \Sigma \alpha: \kappa. \kappa. \text{unit}, \pi_1 \beta, \pi_2 \beta / \alpha, \gamma) \cs
\]

\[P(\text{sig}) = [\delta: \kappa'. c] \quad P(\text{sdecs}) = \gamma: \kappa. cs
\]

\[
P(\text{mod} \triangleright s: \text{sig}, \text{sdecs}) = \beta: (\Sigma \alpha_1: \kappa. \kappa. c. \pi_1 \beta, \pi_2 \beta / \alpha_s, \gamma) \cs
\]

4.2 Modules

\[P(\text{mod}) = m
\]

\[
P(s) = [\alpha_s, x_s]
\]

\[P(\text{sbnds}) = c; x_1 = e_1, \ldots, x_n = e_n
\]

\[
P(\text{sbnds}) = [c, \text{let } x_1 = e_1 \text{ in } \ldots \text{let } x_n = e_n \text{ in } (x_1, \ldots, x_n)]
\]

\[P(\lambda \text{mod}) = [\lambda \alpha_s: \kappa. \lambda x_s: [\alpha_s / \alpha] c] e
\]

\[P(\text{mod} \triangleright s: \text{sig}, \text{mod}_1) = [\lambda \alpha_s: \kappa. \lambda x_s: [\alpha_s / \alpha] c] e
\]

\[P(\text{mod} mod_1) = [c', e] (\alpha / \alpha)
\]

\[P(\text{mod} mod_1) = [c', e']
\]

\[P(\text{mod} mod_1) = [c', e']
\]

9
\[ P(\text{mod}_c) = [c, e] \quad \text{lab} = \text{lab}_i \]
\[ P(\text{mod}_{\text{lab}_1, \ldots, \text{lab}_n}) = [\pi', \pi, e] \]
\[ P(\text{mod}) = m \]
\[ P(\text{mod} : \triangleright \text{sig}) = m \]
\[ P(\text{let } s = \text{mod } \text{in } (\text{mod}' : \text{sig}')) = [[c/\alpha_s]c', \text{let } x_s = e \text{ in } [c/\alpha_s]e'] \]
\[ P(\text{mod}) = P(\text{mod}') = [c', e'] \]
\[ P(\text{let } s = \text{mod } \text{in } (\text{mod}' : \text{sig}')) = [[c/\alpha_s]c', \text{let } x_s = e \text{ in } [c/\alpha_s]e'] \]

\[ P(\text{sbnds}) = c; \text{ebnds} \]
\[ P() = \ast; \cdot \]
\[ P(\text{exp}) = e \quad P(\text{sbnds}) = c; \text{ebnds} \]
\[ P(\text{con}) = c' \quad P(\text{sbnds}) = c; \text{ebnds} \]
\[ P(\text{lab} \triangleright x = \text{exp}, \text{sbnds}) = (\ast, c); x = e, \text{ebnds} \]
\[ P(\text{lab} \triangleright \alpha = \text{con}, \text{sbnds}) = [c', [c'/\alpha]c]; x_s = e', [c'/\alpha]e'_s \]  

Rule (16): A con has no dynamic part so it is okay to just make up a fresh variable for the ebnds part.
\[ P(\text{mod}) = [c', e'] \quad P(\text{sbnds}) = c; \text{ebnds} \]
\[ P(\text{lab} \triangleright s = \text{mod}, \text{sbnds}) = [c', [c'/\alpha]c]; x_s = e', [c'/\alpha]e'_s \]

4.3 Kinds
\[ P(\text{knd}) = \kappa \]
\[ P(\text{T}) = \kappa \]
\[ P(\text{con}) = c \]
\[ P(\text{SBnds}) = c; \text{ebnds} \]
\[ P(\text{lab} \triangleright \text{in } (\text{mod}' : \text{sig}')) = [[c/\alpha_s]c', \text{let } x_s = e \text{ in } [c/\alpha_s]e'] \]

4.4 Constructors
\[ P(\text{con}) = c \]
\[ P(\text{lab} \triangleright \text{in } (\text{mod}' : \text{sig}')) = [[c/\alpha_s]c', \text{let } x_s = e \text{ in } [c/\alpha_s]e'] \]

Rule (16): A con has no dynamic part so it is okay to just make up a fresh variable for the ebnds part.
\[ P(\text{mod}) = [c', e'] \quad P(\text{sbnds}) = c; \text{ebnds} \]
\[ P(\text{lab} \triangleright s = \text{mod}, \text{sbnds}) = [c', [c'/\alpha]c]; x_s = e', [c'/\alpha]e'_s \]

4.4 Constructors
\[ P(\text{con}) = c \]
\[ P(\text{lab} \triangleright \text{in } (\text{mod}' : \text{sig}')) = [[c/\alpha_s]c', \text{let } x_s = e \text{ in } [c/\alpha_s]e'] \]

Rule (16): A con has no dynamic part so it is okay to just make up a fresh variable for the ebnds part.
\[ P(\text{knd}) = \kappa \quad P(\text{con}) = c \]
\[ P(\mu: \text{knd}.\text{con}) = \mu: \kappa.c \]
\[ P(\text{mod}_n) = [c, e] \quad \text{lab} = \text{lab}_i \]
\[ P(\text{mod}_n^{\text{lab}_1, \ldots, \text{lab}_n}.\text{lab}_j) = \pi^n c \]
\[ P(\text{int}) = \text{int} \]
\[ P(\text{char}) = \text{char} \]
\[ P(\text{string}) = \text{string} \]
\[ P(\text{con}_1) = c_1 \quad P(\text{con}_2) = c_2 \]
\[ P(\text{con}_1 \rightarrow \text{con}_2) = c_1 \rightarrow c_2 \]
\[ P(\text{con}) = c \]
\[ P(\text{tag con}) = \text{tag} c \]
\[ P(\text{con}_i) = c_i \quad \text{forall } 1 \leq i \leq n \]
\[ P([\text{lab}_1: \text{con}_1, \ldots, \text{lab}_n: \text{con}_n]) = [c_1, \ldots, c_n] \]
\[ P(\times [\text{lab}_1: \text{con}_1, \ldots, \text{lab}_n: \text{con}_n]) = \times [c_1, \ldots, c_n] \]

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

\[ P(\text{knd}) = \kappa \quad P(\text{con}) = c \]
\[ P(\forall \alpha: \text{knd}.\text{con}) = \forall \alpha: \kappa.c \]

### 4.5 Expressions

\[ P(\text{exp}) = e \]

\[ P(\text{x}) = \text{x} \]
\[ P(\pi) = \pi \]
\[ P(\text{char}) = \text{char} \]
\[ P(\text{string}) = \text{string} \]
\[ P(\text{con}_i) = c_i \quad \text{forall } 1 \leq i \leq n \]
\[ P(\text{fix} [f_i (x_i; \text{con}_i) : \text{con}'_i.\text{exp}_i]_{i=1}^n \text{ end}) = \text{fix} [f_i (x_i; c_i) : c'_i.e_i]_{i=1}^n \text{ end} \]
\[ P(\text{exp}_1) = e_1 \quad P(\text{exp}_2) = e_2 \]
\[ P(\text{exp}_1 \text{ binop } \text{exp}_2) = e_1 \text{ binop } e_2 \]
\[ P(\text{unop } \text{exp}) = \text{unop } e \]
\[ P(\text{exp}_1) = e_1 \quad P(\text{exp}_2) = e_2 \]
\[ P(\text{exp}_1 \text{ exp}_2) = e_1 e_2 \]
\[ P(\text{exp}_1) = e_1 \quad P(\text{exp}_2) = e_2 \]

\[ P(\text{handle \ exp}_1 \text{ with \ exp}_2) = \text{handle} \ e_1 \text{ with } e_2 \]  \hspace{1cm} (47)

\[ P(\text{con}) = c \quad P(\text{exp}) = e \]

\[ P(\text{raise}^\text{con} \ \text{exp}) = \text{raise}^c e \]  \hspace{1cm} (48)

\[ P(\text{ref \ exp}) = \text{ref } e \]

\[ P(\text{exp}) = e \]

\[ P(\text{get \ exp}) = \text{get } e \]  \hspace{1cm} (50)

\[ P(\text{exp}_1) = e \quad P(\text{exp}_2) = e_2 \]

\[ P(\text{set}(\text{exp}_1, \text{exp}_2)) = \text{set}(e_1, e_2) \]  \hspace{1cm} (51)

\[ P(\text{con}) = c \quad P(\text{exp}) = e \]

\[ P(\text{roll}^\text{con} \ \text{exp}) = \text{roll}^c e \]  \hspace{1cm} (52)

\[ P(\text{exp}) = e \]

\[ P(\text{unroll \ exp}) = \text{unroll } e \]  \hspace{1cm} (53)

\[ P(\text{exp}_i) = e_i \quad \text{forall } 1 \leq i \leq n \]

\[ P((\text{lab}_1 \mapsto \text{exp}_1, \ldots, \text{lab}_n \mapsto \text{exp}_n)) = (e_1, \ldots, e_n) \]  \hspace{1cm} (54)

\[ P(\text{lab} = \text{lab}) \]

\[ P(\text{con}, \ldots, \text{lab}_n) \]

\[ P(\text{ref} \ \text{con}, \ldots, \text{lab}_n \ \text{exp}) = \text{ref} \ e \]  \hspace{1cm} (55)

\[ P(\text{prop}(\text{lab}_i, \text{lab}_n)) = \text{prop}(e_i, e_n) \]  \hspace{1cm} (56)

\[ P(\text{case}^\text{con} \ \text{exp \ of} \ \text{lab}_1 \rightarrow \text{exp}_1, \ldots, \text{lab}_n \rightarrow \text{exp}_n) = \text{case}^c e \text{ of } e_1, \ldots, e_n \]  \hspace{1cm} (57)

\[ P(\text{tag}(\text{exp}_1, \text{exp}_2)) = \text{tag}(e_1, e_2) \]  \hspace{1cm} (58)

\[ P(\text{con}) = c \]

\[ P(\text{newtag}^{\text{con}}) \]

\[ P(\text{newtag}[\text{con}]) = \text{newtag}[c] \]  \hspace{1cm} (59)

\[ P(\text{exp}_1) = e_1 \quad P(\text{exp}_2) = e_2 \quad P(\text{exp}_4) = e_4 \quad P(\text{exp}_3) = e_3 \]

\[ P(\text{iftag} \ \text{of} \ \text{exp}_1 \ \text{is} \ \text{exp}_2 \ \text{else} \ \text{exp}_4) = \text{iftag} \ e_1 \text{ is } e_2 \text{ then } e_3 \text{ else } e_4 \]  \hspace{1cm} (60)

\[ P(\text{ref} \ \text{con}) = \kappa \]

\[ P(\Lambda \ast \text{con} \ \text{exp}) = \text{ref} \ k \text{ of } e \]  \hspace{1cm} (61)

\[ P(\text{exp}) = e \quad P(\text{con}) = c \]

\[ P(\text{esp}[\text{con}]) = e[c] \]  \hspace{1cm} (62)

\[ P(\text{let} \ x = \text{exp} \ \text{in} \ \text{exp}') = \text{let} \ x = e \ \text{in} \ e' \]  \hspace{1cm} (63)

\[ P(\text{exp}) = e \quad P(\text{mod}) = [c', e'] \]

\[ P(\text{mod} \ \text{of} \ \text{in} \ \text{exp}) = \text{let} \ x_s = e' \ \text{in} \ [c'/a_s] e \]  \hspace{1cm} (64)

\[ P(\text{mod} \ e) = [c, e] \quad \text{lab} = \text{lab}_i \]

\[ P(\text{mod} \ \text{of} \ \text{lab}_1 \ldots \text{lab}_n) \]  \hspace{1cm} (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 ill/ill1examples.sml.

5.1 Kinds

- PhaseSplitTop.split_kind IL1Examples.kind;
  |- *$(*$[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
]
val it = () : unit