ART Memo 12 (Version 1)
The Redex Trail Transformation

Olaf Chitil

February 20, 2001

Abstract

Here we describe the transformation of a program into a program that computes its redex trail besides its normal result. The description is very close to the actual implementation.

1 General Comments

Everywhere where a runtime error may occur (e.g. no matching pattern) a call to the function fail is inserted by the transformation. Also the standard function error is replaced by primError (in the prelude definition). primError outputs an error message and then connects to the browser. If an exception handler could catch these errors, then the introduction of fail and primError would be unnecessary. Note also that the current scheme would not work anymore if we added such exception handlers to the language.

The computation can only be interrupted by Control-C when a function is entered. The advantage is that the redex trail is in a well-defined state and that computation can be resumed. The latter would hardly be possible, if Control-C were caught by an exception handler.

The transformation assumes that a source reference of an expression is only the position where the expressions starts and hence some source references are equal, for example the source references of the expression fx and the f inside are equal. Source references that describe whole blocks would need to be handled with more care in the transformation.

2 The Data Types

$$\text{data R } a = \text{R } a \text{ Trace}$$

$$\text{type Fun } a\ b = \text{Trace} \to \text{R } a \to \text{R } b$$

(In the future this should be introduced as a newtype, so that in types $\to$ can simply be replaced by Fun, even when applied to less than two types.)

$$
\begin{aligned}
\text{data Trace} = {} & \text{Ap Trace [Trace] SR} \\
& \mid \text{Nm Trace NmType SR} \\
& \mid \text{Ind Trace Trace} \\
& \mid \text{Root} \\
& \mid \text{Sat Trace Trace} \\
& \mid \text{Pruned} \\
& \mid \text{Hidden Trace}
\end{aligned}
$$

3 Trusting

$\text{trust} :: \text{Trace} \to \text{Bool}$

$\text{trust}\ t = \text{trustFun}\ t$

$\text{trusted} :: \text{Trace} \to \text{Trace} \to \text{Bool}$

$\text{trusted}\ t\ \mathit{tf} = \text{trustFun}\ \mathit{tf}\ \ \&\&\ \text{trustFun}\ t$

$\text{trustFun} :: \text{Trace} \to \text{Bool}$

$$
\begin{aligned}
&\text{trustFun (Ap \_ } (\mathit{tf} : \_)\ \_) = \text{trustFun}\ \mathit{tf} \\
&\text{trustFun (Nm \_ (NTTrusted \_) \_)} = \text{True} \\
&\text{trustFun (Nm \_ (NTId \_) \_)} = \text{False} \\
&\text{trustFun (Nm \_ NTLambda \_)} = \text{False} \quad \text{definitely wrong, should follow trace} \\
&\text{trustFun (Nm } t \text{ NTCase \_)} = \text{trustFun}\ t \\
&\text{trustFun (Nm } t \text{ NTIf \_)} = \text{trustFun}\ t \\
&\text{trustFun (Nm } t \text{ NTGuard \_)} = \text{trustFun}\ t \\
&\text{trustFun (Sat } t\ \_) = \text{trustFun}\ t \\
&\text{trustFun (Ind \_ } t) = \text{trustFun}\ t \\
&\text{trustFun (Root)} = \text{False} \\
&\text{trustFun (Pruned)} = \text{False} \\
&\text{trustFun (Hidden \_)} = \text{True}
\end{aligned}
$$

The functions trust, trusted and trustFun are actually primitive C-function (to improve performance?). trusted also increases a reduction counter (but never the counter from calls of suspected functions from trusted functions, because this should not happen; but it certainly does for higher-order functions).

4 Transformation of a Pattern

$$
\begin{aligned}
\mathcal{P}[\![C\ p_1\ \dots\ p_i]\!] &= \text{R}\ (C\ \mathcal{P}[\![p_1]\!]\ \dots\ \mathcal{P}[\![p_i]\!]) \\
\mathcal{P}[\![x]\!] &= x \\
\mathcal{P}[\![\_]\!] &= \_ \\
\mathcal{P}[\![x@p]\!] &= x@\mathcal{P}[\![p]\!] \\
\mathcal{P}[\![\sim p]\!] &= \sim \mathcal{P}[\![p]\!]
\end{aligned}
$$

String literals are transformed into lists of characters. Integer and Rational literals are wrapped by dummy functions which are removed after type checking.

5 Transformation of Expressions

The boolean argument states if the expression is in a non-projective context, that is, it is False if the expression is the result of the function in whose rhs it appears (not simply the rhs because of the case and if construct). It is used both for introduction of indirection nodes and for trusting.

Superscripts at expressions (e.g. M^s) are source references.

5.1 Variable

$x, x_1, \ldots$ are λ-bound variables. $y, y_1, \ldots$ are let-bound variables.

$$\begin{aligned}
\mathcal{E}[\![x]\!]_t^{\text{True}} &= x \\
\mathcal{E}[\![x]\!]_t^{\text{False}} &= \text{indir } t\ x \\
\mathcal{E}[\![y^s]\!]_t &= y\ s\ t
\end{aligned}$$

$\text{indir} :: \text{Trace} \to \text{R}\ a \to \text{R}\ a$

$\text{indir}\ t\ (\text{R}\ v\ t') = \text{R}\ v\ (\text{Ind}\ t\ t')$

5.2 Data Constructor Application

$$\mathcal{E}[\![(C\ M_1 \ldots M_i)^s]\!]_t = \begin{cases} \text{con}_i\ s\ t\ C\ \mathcal{N}[\![C]\!]\ \mathcal{E}[\![M_1]\!]_t^{\text{True}} \ldots \mathcal{E}[\![M_i]\!]_t^{\text{True}} & \text{, if arity}(C) = i \\ \text{pa}_i\ C\ \text{cn}_{\text{arity}(C)-i}\ s\ t\ \mathcal{N}[\![C]\!]\ \mathcal{E}[\![M_1]\!]_t^{\text{True}} \ldots \mathcal{E}[\![M_i]\!]_t^{\text{True}} & \text{, otherwise} \end{cases}$$

Apply data constructor to wrapped arguments and wrap it with a trace, an application node for the constructor application.

$\text{con}_i :: \text{SR} \to \text{Trace} \to (\text{R}\ a_1 \to \ldots \to \text{R}\ a_n \to b) \to \text{NmType} \to \text{R}\ a_1 \to \ldots \to \text{R}\ a_n \to \text{R}\ b$

$\text{con}_0\ s\ t\ c\ nm = \text{R}\ c\ (\text{Nm}\ t\ nm\ s)$
$\text{con}_i\ s\ t\ c\ nm\ a_1@(\text{R}\ \_\ t_1)\ \ldots\ a_i@(\text{R}\ \_\ t_i) = \text{R}\ (c\ a_1\ \ldots\ a_i)\ (\text{Ap}\ t\ [\text{Nm}\ t\ nm, t_1, \ldots, t_i]\ s)$

Transform a function into a wrapped function with the given trace. For $i > 0$:

$\text{cn}_i :: (R\ a_1 \to \ldots \to R\ a_i \to b) \to \text{Trace} \to (a_1\ \text{`Fun`}\ \ldots\ \text{`Fun`}\ a_i\ \text{`Fun`}\ b)$

$\text{cn}_i\ f\ t = t\ \text{`seq`}\ (\text{R}\ (\lambda t_1\ a_1 \to \ldots \text{R}\ (\lambda t_i\ a_i \to R\ (f\ a_1\ \ldots\ a_i)\ t)\ t \ldots)\ t)$

Create application node for function that is partially applied to i arguments and transform partial application with given function.

$\text{pa}_i :: (\text{R}\ a_1 \to \ldots \to \text{R}\ a_i \to b) \to (b \to \text{Trace} \to c) \to \text{SR} \to \text{Trace} \to \text{NmType}$
$\quad \to \text{R}\ a_1 \to \ldots \to \text{R}\ a_i \to c$

$\text{pa}_0\ cn\ c\ s\ t\ nm = cn\ c\ (\text{Nm}\ t\ nm\ s)$
$\text{pa}_i\ cn\ c\ s\ t\ nm\ a_1@(\text{R}\ \_\ t_1)\ \ldots\ a_i@(\text{R}\ \_\ t_i)$
$\quad = cn\ (c\ a_1 \ldots a_i)\ (\text{Ap}\ t\ [\text{Nm}\ t\ nm\ s, t_1, \ldots, t_i]\ s)$

5.3 Other Application

$$\mathcal{E}[\![M\ N_1\ \ldots\ N_i]\!]_t^{\text{True}} = \text{ap}_i\ s\ t\ \mathcal{E}[\![M_1]\!]_t^{\text{True}}\ \ldots\ \mathcal{E}[\![M_i]\!]_t^{\text{True}}$$
$$\mathcal{E}[\![M\ N_1\ \ldots\ N_i]\!]_t^{\text{False}} = \text{rap}_i\ s\ t\ \mathcal{E}[\![M_1]\!]_t^{\text{True}}\ \ldots\ \mathcal{E}[\![M_i]\!]_t^{\text{True}}$$

If a trusted function is applied in a trusted context, then the application is not registered in the trail. In that case ap_i also marks the trace as hidden. In contrast, rap_i does not do the latter. Is the trace already marked as hidden?

rap_i is used within a projective context, otherwise ap_i is used.

$\text{ap}_i :: \text{SR} \rightarrow \text{Trace} \rightarrow \text{R}\ (a_1\ \text{`Fun`}\ \ldots\ \text{`Fun`}\ a_i\ \text{`Fun`}\ b) \rightarrow \text{R}\ a_1 \rightarrow \ldots \rightarrow \text{R}\ a_i \rightarrow R\ b$

$\text{ap}_1\ s\ t\ (\text{R}\ rf\ tf)\ a_1@(\text{R}\ \_\ t_1) =$
$\quad \text{if trusted}\ t\ tf$
$\quad\quad \text{then let}\ t' = \text{hidden}\ t\ \text{in}\ t'\ \text{`seq`}\ rf\ t'\ a_1$
$\quad\quad \text{else let}\ t' = \text{Ap}\ t\ [tf, t_1]\ s\ \text{in}\ rf\ t'\ a_1$

$\text{ap}_i\ s\ t\ (\text{R}\ rf\ tf)\ a_1@(\text{R}\ \_\ t_1)\ \ldots\ a_i@(\text{R}\ \_\ t_i) =$
$\quad \text{if trusted}\ t\ tf$
$\quad\quad \text{then let}\ t' = \text{hidden}\ t\ \text{in}\ t'\ \text{`seq`}\ \text{pap}_{i-1}\ s\ t'\ (rf\ t'\ a_1)\ a_2\ \ldots\ a_i$
$\quad\quad \text{else let}\ t' = \text{Ap}\ t\ [tf, t_1, \ldots, t_i]\ s\ \text{in}\ \text{pap}_{i-1}\ s\ t'\ (rf\ t'\ a_1)\ a_2\ \ldots\ a_i$

$\text{hidden} :: \text{Trace} \rightarrow \text{Trace}$

$\text{hidden}\ h@(\text{Hidden}\ \_) = h$
$\text{hidden}\ t = \text{Hidden}\ t$

$\text{rap}_i :: \text{SR} \rightarrow \text{Trace} \rightarrow \text{R}\ (a_1\ \text{`Fun`}\ \ldots\ \text{`Fun`}\ a_i\ \text{`Fun`}\ b) \rightarrow \text{R}\ a_1 \rightarrow \ldots \rightarrow \text{R}\ a_i \rightarrow \text{R}\ b$

$\text{rap}_1\ s\ t\ (\text{R}\ rf\ tf)\ a_1@(\text{R}\ \_\ t_1) =$
$\quad \text{if trusted}\ t\ tf$
$\quad\quad \text{then}\ rf\ t\ a_1$
$\quad\quad \text{else let}\ t' = \text{Ap}\ t\ [tf, t_1]\ s\ \text{in}\ rf\ t'\ a_1$

$\text{rap}_i\ s\ t\ (\text{R}\ rf\ tf)\ a_1@(\text{R}\ \_\ t_1)\ \ldots\ a_i@(\text{R}\ \_\ t_i) =$
$\quad \text{if trusted}\ t\ tf$
$\quad\quad \text{then}\ \text{pap}_{i-1}\ s\ t\ (rf\ t\ a_1)\ a_2\ \ldots\ a_i$
$\quad\quad \text{else let}\ t' = \text{Ap}\ t\ [tf, t_1, \ldots, t_i]\ s\ \text{in}\ \text{pap}_{i-1}\ s\ t'\ (rf\ t'\ a_1)\ a_2\ \ldots\ a_i$

$i \geq 0$:

$\text{pap}_i :: \text{SR} \rightarrow \text{Trace} \rightarrow \text{R}\ (a_1\ \text{`Fun`}\ \ldots\ \text{`Fun`}\ a_i\ \text{`Fun`}\ b) \rightarrow \text{R}\ a_1 \rightarrow \ldots \rightarrow \text{R}\ a_i \rightarrow R\ b$

$\text{pap}_0\ s\ t\ e = e$

$\text{pap}_i\ s\ t\ (\text{R}\ rf\ tf)\ a_1@(\text{R}\ \_\ t_1)\ \ldots\ a_i@(\text{R}\ \_\ t_i) =$
$\quad \text{if}\ t\ \text{`sameAs`}\ tf$
$\quad\quad \text{then}\ \text{pap}_{i-1}\ s\ t\ (rf\ t\ a_1)\ a_2\ \ldots\ a_i$
$\quad\quad \text{else let}\ t' = \text{Ap}\ t\ [tf, t_1, \ldots, t_i]\ s\ \text{in}\ \text{pap}_{i-1}\ s\ t'\ (rf\ t'\ a_1)\ a_2\ \ldots\ a_i$

5.4 Lambda Abstraction

$$\mathcal{E}[\![(\lambda x_1 \ldots x_i.M)^s]\!]_t = \mathrm{fun}_i\ \mathcal{N}[\![\lambda]\!]\ (\lambda t' x_1 \ldots x_n.\mathcal{E}[\![M]\!]_{t'}^{\mathrm{False}})\ s\ t$$

Actually the translated λ-abstraction uses a case-construct to test if the patterns match. Otherwise it calls a function that outputs an error message and directly connects to the browser.

$$\begin{aligned}\mathrm{fun}_i &:: \mathrm{NmType} \to (\mathrm{Trace} \to \mathrm{R}\ a_1 \to \ldots \to \mathrm{R}\ a_i \to \mathrm{R}\ b) \to SR \to \mathrm{Trace}\\ &\quad \to \mathrm{R}(a_1\ \text{`Fun`}\ \ldots\ \text{`Fun`}\ a_i\ \text{`Fun`}\ b)\end{aligned}$$

$$\begin{aligned}
&\mathrm{fun}_0\ nm\ rf\ s\ t = \mathrm{let}\ t' = \mathrm{Nm}\ t\ nm\ s\\
&\qquad\qquad\qquad\qquad\quad \mathrm{R}\ r\ rt = \mathrm{enter}\ nm\ t'\ (rf\ t)\\
&\qquad\qquad\qquad\quad \mathrm{in}\ \mathrm{R}\ r\ (\mathrm{Sat}\ t'\ rt)\\
&\mathrm{fun}_i\ nm\ rf\ s\ t = \mathrm{R}\ (\lambda t_1 a_1 \to \ldots \mathrm{R}\ (\lambda t_i a_i \to \mathrm{let}\ \mathrm{R}\ r\ rt = \mathrm{enter}\ nm\ t_i\ (rf\ t_i\ a_1\ \ldots\ a_i)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{in}\ \mathrm{R}\ r\ (\mathrm{Sat}\ t_i\ rt))\ t_{i-1}\\
&\qquad\qquad\qquad\qquad \ldots t_1)\\
&\qquad\qquad\qquad\qquad (\mathrm{Nm}\ t\ nm\ s)
\end{aligned}$$

The primitive C-function enter connects to the redex trail browser, if the user pressed Control-C, execution is in single-stepping mode, etc. Otherwise it just returns the third argument.

5.5 Let Expression

$$\mathcal{E}[\![\mathrm{let}\ \{decl_1; \ldots ; decl_i\}\ \mathrm{in}\ M]\!]_t = \mathrm{let}\ \{\mathcal{D}[\![decl_1]\!]_t; \ldots ; \mathcal{D}[\![decl_i]\!]_t\}\ \mathrm{in}\ \mathcal{E}[\![M]\!]_t^{\mathrm{False}}$$

5.6 Case Expression

$$\mathcal{E}[\![(\mathrm{case}\ M\ \mathrm{of}\ \{alt_1^{\to}; \ldots ; alt_i^{\to}\})^s]\!]_t^b = \mathrm{ap}_1\ s\ t\ (\mathrm{fun}_1\ \mathcal{N}[\![\mathrm{case}]\!]\ (\mathrm{let}\ f\ alts;\ defs\ \mathrm{in}\ f)\ s\ t)\ \mathcal{E}[\![M]\!]_t^b$$

where

$$(alts, defs) = \mathcal{F}[\![alt_1^{=}, \ldots , alt_i^{=}]\!]$$

5.7 If Expression

$$\mathcal{E}[\![\mathrm{if}\ M\ \mathrm{then}\ N_1\ \mathrm{else}\ N_2]\!]_t^b = \mathrm{tif}\ s\ \mathcal{E}[\![M]\!]_t^b\ (\lambda t' \to \mathcal{E}[\![N_1]\!]_{t'}^{\mathrm{False}})\ (\lambda t' \to \mathcal{E}[\![N_2]\!]_{t'}^{\mathrm{False}})\ t$$

$$\mathrm{tif} :: \mathrm{SR} \to \mathrm{R\ Bool} \to (\mathrm{Trace} \to a) \to (\mathrm{Trace} \to a) \to \mathrm{Trace} \to a$$

$$\begin{aligned}
&\mathrm{tif}\ s\ (\mathrm{R}\ iv\ it)\ a_1\ a_2\ t = \mathrm{if\ trust}\ t\\
&\qquad\qquad\qquad\qquad\qquad \mathrm{then\ if}\ iv\ \mathrm{then}\ a_1\ t\ \mathrm{else}\ a_2\ t\\
&\qquad\qquad\qquad\qquad\qquad \mathrm{else\ let}\ t' = \mathrm{Ap}\ t\ [\mathrm{Nm}\ t\ \mathcal{N}[\![\mathrm{if}]\!]\ s, it, t]\ s\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathrm{in\ if}\ iv\ \mathrm{then}\ a_1\ t'\ \mathrm{else}\ a_2\ t'
\end{aligned}$$

6 Transformation of Declarations

Bindings in class and instance definitions are basically function bindings. A few special cases are however treated specially.

6.1 Function binding

Let i be the arity of the definition of the function f.

$$\begin{array}{l}\mathcal{D}[\![f\ \mathit{alts}]\!] = (f\ \ s\ t = \mathrm{fun}_i\ \mathcal{N}[\![f]\!]\ f'\ \ s\ t\ \text{where}\ f'\ \mathit{alts}';\ \mathit{defs}) \\ \qquad \text{where} \\ \qquad (\mathit{alts}', \mathit{defs}) = \mathcal{F}[\![\mathit{alts}]\!]\end{array}$$

Transformation of binding alternatives:

$$\mathcal{F}[\![\ ]\!] = (t\ \_1 \ldots \_i = \mathrm{fail}\ t,\ )$$

$$\begin{array}{l}\mathcal{F}[\![p_1 \ldots p_i\ \ =\ \ M\ \text{where}\ \mathit{decls}; \mathit{alts}]\!] = \\ (t\ \mathcal{P}[\![p_1]\!] \ldots \mathcal{P}[\![p_i]\!]\ \ =\ \ \mathcal{E}[\![M]\!]_t^{\mathrm{False}}\ \text{where}\ \mathcal{D}[\![\mathit{decls}]\!]_t; \mathit{alts}',\ \mathit{defs}) \\ \text{where} \\ (\mathit{alts}', \mathit{defs}) = \mathcal{F}[\![\mathit{alts}]\!]\end{array}$$

$$\begin{array}{l}\mathcal{F}[\![p_1 \ldots p_i\ \mathit{gdes}\ \text{where}\ \mathit{decls}; \mathit{alts}]\!] \\ \mid \text{guards in } \mathit{gdes} \text{ may fail and } \mathit{alts} \text{ not empty} = \\ \quad (t\ \mathcal{P}[\![a_1@p_1]\!] \ldots \mathcal{P}[\![a_1@p_i]\!]\ \ =\ \ \mathcal{G}[\![\mathit{gdes}]\!]_t^{\lambda t.f\ t\ a_1 \ldots a_i}\ t\ \text{where}\ \mathcal{D}[\![\mathit{decls}]\!]_t \\ \quad ; t\ a_1 \ldots a_i = f\ t\ a_1 \ldots a_i,\ \ f\ \mathit{alts}';\ \mathit{defs}) \\ \quad \text{where} \\ \quad (\mathit{alts}', \mathit{defs}) = \mathcal{F}[\![\mathit{alts}]\!] \\ \mid \text{otherwise} = \\ \quad (t\ \mathcal{P}[\![a_1@p_1]\!] \ldots \mathcal{P}[\![a_1@p_i]\!]\ \ =\ \ \mathcal{G}[\![\mathit{gdes}]\!]_t^{\mathrm{fail}}\ t\ \text{where}\ \mathcal{D}[\![\mathit{decls}]\!]_t;\ \mathit{alts}',\ \mathit{defs}) \\ \quad \text{where}(\mathit{alts}', \mathit{defs}) = \mathcal{F}[\![\mathit{alts}]\!]\end{array}$$

Transformation of guarded expressions:

$$\begin{array}{c}\mathcal{G}[\![\ ]\!]_t^F = \lambda t'.\,F\ t' \\ \mathcal{G}[\![\ \mid N^s\ \ =\ \ M; \mathit{gdes}]\!]_t^F = \mathrm{t\_guard}\ s\ \mathcal{E}[\![N]\!]_t^{\mathrm{False}}\ (\lambda t'.\mathcal{E}[\![M]\!]_{t'}^{\mathrm{False}})\ \mathcal{G}[\![\mathit{gdes}]\!]_t^F\end{array}$$

Just as for if, except for the different name.

$$\mathrm{t\_guard} :: \mathrm{SR} \to \mathrm{R\ Bool} \to (\mathrm{Trace} \to a) \to (\mathrm{Trace} \to a) \to \mathrm{Trace} \to a$$

$$\begin{array}{ll}\mathrm{t\_guard}\ s\ (\mathrm{R}\ gv\ gt)\ e\ \mathit{cont}\ t & = \text{if trust}\ t \\ & \quad \text{then if}\ gv\ \text{then}\ e\ t\ \text{else}\ \mathit{cont}\ t \\ & \quad \text{else let}\ t'\ =\ \mathrm{Ap}\ t\ [\mathrm{Nm}\ t\ \mathcal{N}[\![\mathrm{guard}]\!]\ s, gt, t]\ s \\ & \qquad \text{in if}\ gv\ \text{then}\ e\ t'\ \text{else}\ \mathit{cont}\ t'\end{array}$$

6.2 Constant binding

A constant binding is transformed like a nullary function binding, except that sharing is preserved and hence the parent relationship is rather useless. The transformation is more simple, because a constant binding consists only of a single alternative.

$$
\begin{aligned}
\mathcal{D}[\![x^s\ rhs \text{ where } decls]\!]_t = (&x\ \_\ \_ = x' \\
&; x' = \text{lazySat}\ \mathcal{R}[\![rhs]\!]_{t'}\ t' \\
&\quad \text{where} \\
&\quad t' = \text{Nm}\ t\ \mathcal{N}[\![x]\!]\ s \\
&\quad \mathcal{D}[\![decls]\!]_{t'})
\end{aligned}
$$

Transform right hand side.

$$
\begin{aligned}
\mathcal{R}[\![M]\!]_t &= \mathcal{E}[\![M]\!]_t^{\text{False}} \\
\mathcal{R}[\![gdes]\!]_t &= \mathcal{G}[\![gdes]\!]_t^{\text{fail}}\ t
\end{aligned}
$$

Assure that a trace component of wrapped value exists by construction of Sat. Similar to fun_0.

$$
\begin{aligned}
&\text{lazySat} :: \text{R}\ a \to \text{Trace} \to \text{R}\ a \\
&\text{lazySat}\ a\ t\ =\ \text{let R}\ v\ vt\ =\ a \text{ in R}\ v\ (\text{Sat}\ t\ vt)
\end{aligned}
$$

6.3 Pattern binding

Unlike all other let-defined variables, variables defined through pattern bindings are not registered in the trace. A pattern binding adds no node to the trace which is bad in practise. Note that a pattern binding with just a variable on the left is actually a constant binding; hence the trace construction should actually be similar.

Let $x_1, \ldots, x_i$ be the variables occurring in the pattern p.

$$
\begin{aligned}
\mathcal{D}[\![p\ rhs \text{ where } decls]\!]_t = (&y = \text{case}\ \mathcal{R}[\![rhs]\!]_t \text{ of } \mathcal{P}[\![p[\overline{x}'/\overline{x}]]\!] \to (x'_1, \ldots, x'_i); \_ \to \text{fail}\ t \\
&; x_1\ \_\ \_\ =\ \text{case}\ y \text{ of } (x'_1, \ldots, x'_i) \to x_1 \\
&\ldots \\
&; x_i\ \_\ \_\ =\ \text{case}\ y \text{ of } (x'_1, \ldots, x'_i) \to x_i \\
&)
\end{aligned}
$$

6.4 Foreign Functions

A wrapper is created that unwraps and wraps traces of arguments and result and calls the actual foreign function.

a is the arity of the function.

$$
\begin{aligned}
\mathcal{D}[\![(\text{foreign import}\ cname\ f\ a\ c\ ::\ \tau)^s]\!] = (&\text{foreign import}\ cname\ f'\ a\ c\ ::\ \tau \\
&; f\ s\ t\ =\ \text{prim}_a\ \mathcal{N}[\![f]\!]\ f'\ s\ t \\
&)
\end{aligned}
$$

$$
\begin{array}{l}
\text{prim}_i :: \text{NmCoerce}\ c \Rightarrow \text{NmType} \rightarrow (a_1 \rightarrow \ldots a_i \rightarrow r) \rightarrow \text{SR} \rightarrow \text{Trace} \rightarrow \\
\qquad (a_1\ \text{`Fun`} \ldots \text{`Fun`} a_i \rightarrow r)
\end{array}
$$

$$
\begin{array}{lcl}
\text{prim}_i\ nm\ f\ s\ t & = & \text{let}\ t' \ =\ \text{Nm}\ t\ nm\ s \\
& & \text{in R}\ (\lambda t_1\ (\text{R}\ a_1\ t_1) \rightarrow \\
& & \qquad \ldots \\
& & \qquad\quad \text{R}\ (\lambda t_i\ (\text{R}\ a_i\ t_i) \rightarrow \\
& & \qquad\qquad\quad (\lambda(\text{R}\ v\ vt) \rightarrow \text{R}\ v\ (\text{Sat}\ t_i\ vt)) \\
& & \qquad\qquad\quad (\text{primEnter}\ s\ nm\ t_i\ (f\ a_1 \ldots a_i))) \\
& & \qquad\quad \ldots \\
& & \qquad\quad t_1) \\
& & \qquad t'
\end{array}
$$

Counterpart to 'enter' for primitives: ensures that the result trace is fully evaluated at exactly the same time as the result value.

$$
\text{primEnter} :: \text{NmCoerce}\ a \Rightarrow \text{SR} \rightarrow \text{NmType} \rightarrow \text{Trace} \rightarrow a \rightarrow \text{R}\ a
$$

$$
\begin{array}{lcl}
\text{primEnter}\ s\ nm\ t\ e & = & \text{let}\ v \ =\ \text{enter}\ nm\ t\ e \\
& & \qquad vn \ =\ \text{toNm}\ t\ v\ sr \\
& & \text{in}\ v\ \text{`myseq`}\ vn
\end{array}
$$

The wrapping function toNm is the method of the class NmCoerce.