A Core Language for Extended Pattern Matching and Binding Boolean Expressions

Arthur Charguéraud, Yanni Lefki

Inria Camus & University of Strasbourg, CNRS, ICube

ML Workshop, October 16, 2025

Motivation

Story of pattern matching:

- introduced with ML in the 1970's
- exhaustiveness checking in the 1980's
- efficient compilation scheme in the late 1990's
- first-class patterns in Racket/Scheme the 1990's
- Haskell views and F# active patterns in the mid-2000's

...still ongoing!

- Rust's construct: `if let PAT = EXPR { BODY }`, in RFC from 2014
- *Ultimate Conditional Syntax* by Cheng and Parreaux, OOPSLA'24

This work: aims to provide a yet slightly simpler presentation of a programming language that includes all the desirable features.

Feature focus #1: smart deconstructors

Smart constructors

```
type trm = {
  desc : desc;
  loc : location option; (* from parser *)
  typ : typ option; } (* from typer *)

and desc =
  | Trm_var of var
  | Trm_bool of bool
  | Trm_if of trm * trm * trm
  | Trm_and of trm * trm
  ...

(* To construct a term in a code transformation, we would write e.g.:
  { desc = Trm_if (t1,t2,t3); loc = None; typ = None }.
  Instead, one can use smart constructors such as [trm_if] and [trm_and]. *)

let trm_make (d : desc): trm =
  { desc = d; loc = None; typ = None }

let trm_if (t1 : trm) (t2 : trm) (t3 : trm) : trm =
  trm_make (Trm_if (t1,t2,t3))
let trm_and (t1 : trm) (t2 : trm) : trm =
  trm_make (Trm_and (t1,t2))
```

Benefits of smart constructors

Smart-constructors offer a clean API for building AST terms, based solely on functions, not exposing the constructors.

Say we decide to remove `Trm_and` and encode this construct using a `if`:

```
and desc =
  | Trm_var of var
  | Trm_bool of bool
  | Trm_if of trm * trm * trm
  (* | Trm_and <-- removed *)

(* updated version of the smart constructor for [trm_and] *)
let trm_and (t1 : trm) (t2 : trm) : trm =
  trm_if (t1, t2, trm_bool false)
```

No need to modify the client code, which continues to call `trm_and`.

What about AST deconstruction?

```
(* Client function to recognize terms of the form [t1 && t2 && t3] *)

let trm_and_3_inv (t : trm) : (trm * trm * trm) option =
  begin match t.desc with
  | Trm_and (t1, { desc = Trm_and (t2,t3); _ }) -> Some (t1,t2,t3)
  | _ -> None
  end

(* This client code mentions the constructor [Trm_and] *)
```

Such an explicit manipulation of constructors suffers from:

1. **lack of abstraction**
   → e.g., if we remove `Trm_and`, all the client code needs to change
2. **lack of conciseness**
   → compared with the construction: `trm_and t1 (trm_and t2 t3)`

Smart deconstructors (view/active patterns)

```
(* [REVISITED] Function to recognize terms of the form [t1 && t2 && t3].
  Inside patterns, [trm_and] is short for [Pattern.trm_and] *)
let trm_and_3_inv (t : trm) : (trm * trm * trm) option =
  match t with
  | trm_and t1 (trm_and t2 t3) -> Some (t1,t2,t3)
  | _ -> None

(* Definition of the smart deconstructors *)
let Pattern.trm_if (t : trm) : (trm * trm * trm) option =
  match t.desc with
  | Trm_if t1 t2 t3 -> Some (t1,t2,t3)
  | _ -> None

let Pattern.trm_bool (t : trm) : bool option =
  match t.desc with
  | Trm_bool b -> Some b
  | _ -> None

(* [Pattern.trm_and], assuming [trm_and] is encoded using a conditional. *)
let Pattern.trm_and (t : trm) : (trm * trm) option =
  match t with
  | trm_if t1 t2 (trm_bool false) -> Some (t1,t2)
  | _ -> None
```

Feature focus #2: tests that export bindings

Motivation for binding-boolean-expressions (BBEs)

Example illustrating redundant accesses to a hashtable:

```
match t with
| Some k when Hashtbl.mem tbl k ->
    let v = Hashtbl.get tbl k in
    f v
| None ->
    a_big_expression
```

The algorithmically efficient code is unpleasant:

```
let cont () = (* need to factorize the continuation *)
  a_big_expression in
match t with
| Some k ->
    begin match Hashtbl.get_opt tbl k with
    | Some v -> f v
    | None -> cont()
    end
| None -> cont()
```

BBEs at play in when-clauses and conditionals

Same example, revisited:

```
match t with
| Some k when Hashtbl.get_opt tbl k is Some v -> f v
| None -> a_big_expression
```

The construct "`t is p`" is a *binding-boolean-expression*: it computes a boolean value *and* binds pattern variables in case the pattern matches.

BBEs can appear in conditionals. The code above could be written:

```
if (t is Some k) && (Hashtbl.get_opt tbl k is Some v)
  then f v
  else a_big_expression
```

Related work: example taken from Rust RFC 2497-2025-06-18.

```
if let Some((fn_name, aft_name)) = s.split_once("(")
  && !fn_name.is_empty()
  && is_legal_ident(fn_name)
  && let Some((args_str, "")) = aft_name.rsplit_once(")") {
   return fn_name + args_str;
}
```

BBEs at play in while loops

The standard pattern:

```
while not (Queue.empty q) do
  let x = Queue.pop q in
  ...
done
```

becomes:

```
while Queue.pop_opt q is Some x do
  ...
done
```

Merge-sort pattern:

```
while Stack.top_opt s1 is Some x1
   && Stack.top_opt s2 is Some x2 do
  if x1 <= x2 then begin
    ignore (Stack.pop s1);
    Stack.push qr x1;
  end else begin
    ignore (Stack.pop s2);
    Stack.push qr x2;
  end
done;
assert (Stack.empty s1 || Stack.empty s2);
```

Formalization

Syntax of terms

A λ-calculus with a switch construct.

$$
\begin{array}{rcll}
t, f, g & := & \mid\ x & \text{variable} \\
 & & \mid\ \lambda(x_1, ..., x_n).t & \lambda\text{-abstraction} \\
 & & \mid\ f\ (t_1, ..., t_n) & \text{application} \\
 & & \mid\ \mathbf{let}\, x\, =\, t_1\, \mathbf{in}\, t_2 & \text{let-binding} \\
 & & \mid\ \mathbf{switch}\, c_1 \mid ... \mid c_n & \text{branching} \\
 & & & \\
c & := & \mid\ \textbf{case}\ b\ \textbf{then}\ t & \text{branch of a } \mathbf{switch}
\end{array}
$$

where b is a *binding-boolean expression*.

Syntax of encoded term constructs

Encoding for **match**:

$$\begin{array}{l}\textbf{match}\, t\, \textbf{with} \\ \quad \mid p_1 \to\ t_1 \\ \quad \mid p_2 \to\ t_2\end{array} \quad \Longrightarrow \quad \begin{array}{l}\textbf{let}\, x\ =\ t\, \textbf{in} \\ \textbf{switch} \\ \quad \mid \textbf{case}\, (x\, \textbf{is}\, p_1)\, \textbf{then}\, t_1 \\ \quad \mid \textbf{case}\, (x\, \textbf{is}\, p_2)\, \textbf{then}\, t_2\end{array}$$

Encoding for **if**, where b binds variables in t_1:

$$\textbf{if}\, b\, \textbf{then}\, t_1\, \textbf{else}\, t_2 \quad \Longrightarrow \quad \begin{array}{l}\textbf{switch} \\ \quad \mid \textbf{case}\, b\, \textbf{then}\, t_1 \\ \quad \mid \textbf{case}\, \mathsf{true}\, \textbf{then}\, t_2\end{array}$$

Note: due to when-clauses and possibly-impure smart deconstructors, subpatterns may have side effects, hence the evaluation order matters a lot.

Syntax of binding-boolean-expressions

b :=	|	t **is** p	pattern-matching	binds $\mathcal{B}(p)$
	|	b_1 **and** b_2	conjunction (same as b_1 `&&` b_2) with b_1 binding into b_2	binds $\mathcal{B}(b_1) \cup \mathcal{B}(b_2)$
	|	b_1 **or** b_2	disjunction (same as b_1 `||` b_2)	binds $\mathcal{B}(b_1) \cap \mathcal{B}(b_2)$
	|	**not** b	negation	binds $\varnothing$

where p is a *pattern*, and where $\mathcal{B}$ means "bindings exported by".

Note: in the paper, we describe a core construct named $\mathbf{switch}^{\text{bbe}}$ that suffices to encode the **and**, **or** and **not** constructs.

Syntax of patterns

p :=	$x^?$	pattern variable	binds $\{x\}$
|	$p_1 \mid p_2$	pattern disjunction	binds $\mathcal{B}(p_1) \cap \mathcal{B}(p_2)$
|	$p_1 \ \& \ p_2$	pattern intersection	binds $\mathcal{B}(p_1) \cup \mathcal{B}(p_2)$
|	p **when** b	guarded pattern	binds $\mathcal{B}(p) \cup \mathcal{B}(b)$
|	$f\ (p_1, ..., p_n)$	inversor pattern	binds $\bigcup_i \mathcal{B}(p_i)$
|	g	predicate pattern with p binding into b	binds $\varnothing$

where :

- f is a smart deconstructor of type: $T \to (T_1, ..., T_n)$ option.
  $f\ (p_1, ..., p_n)$ is equivalent to $x^?$ **when** $f\ (x)$ **is** Some $(p_1, ..., p_n)$.
- classic data constructors can be viewed as smart deconstructors
- g is a boolean function of type: $T \to$ bool.
  g can be viewed as a particular case of a smart deconstructor.

Typing & Semantics

Typing judgments, Typing rules for terms and BBEs

- $E \vdash_{\mathsf{trm}} t : T$
- $E \vdash_{\mathsf{bbe}} b \rightsquigarrow B$
- $E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B$

- E maps context variables to types
- B maps exported variables to types
- Not checking for exhaustiveness

$$\frac{\forall i.\ E \vdash_{\mathsf{trm}} c_i : T}{E \vdash_{\mathsf{trm}} \textbf{switch}\ c_1 \mid \ldots \mid c_n\ :\ T} \qquad \frac{E \vdash_{\mathsf{bbe}} b \rightsquigarrow B \qquad E; B \vdash_{\mathsf{trm}} t : T}{E \vdash_{\mathsf{trm}} \textbf{case}\ b\ \textbf{then}\ t\ :\ T}$$

$$\frac{E \vdash_{\mathsf{trm}} t : T \qquad E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B}{E \vdash_{\mathsf{bbe}} t\ \textbf{is}\ p\ \rightsquigarrow\ B}$$

$$\frac{E \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\mathsf{bbe}} b_1\ \textbf{and}\ b_2\ \rightsquigarrow\ B_1 \uplus B_2} \qquad \frac{E \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B \qquad E \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B}{E \vdash_{\mathsf{bbe}} b_1\ \textbf{or}\ b_2\ \rightsquigarrow\ B}$$

where $B_1 \# B_2$ asserts disjointness. We disallow shadowing in **and**.

The restrict construct

$$\frac{E \vdash_{\mathsf{bbe}} b_1 \rightsquigarrow B \qquad E \vdash_{\mathsf{bbe}} b_2 \rightsquigarrow B}{E \vdash_{\mathsf{bbe}} b_1 \textbf{ or } b_2 \rightsquigarrow B} \qquad \frac{E \vdash_{\mathsf{bbe}} b \rightsquigarrow B \qquad V \subseteq \mathsf{dom}\, B}{E \vdash_{\mathsf{bbe}} (\textbf{restrict}\, V\, b) \rightsquigarrow B_{|V}}$$

The language construct **restrict** $V\, b$ reduces to V the set of bindings that are exported by the BBE b.

Occurrence of **restrict** can be automatically inserted during algorithmic typechecking: b_1 **or** b_2 elaborates to (**restrict** $V\, b_1$) **or** (**restrict** $V\, b_2$), where V denotes the set $\mathcal{B}(b_1) \cap \mathcal{B}(b_2)$.

Likewise, we have **restrict** $V\, p$ for patterns.

Declarative typechecking rules involving disjunctions can assume that every branch binds the exact same set of variables.

Typing rules for patterns

Recall that $E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B$ means that p exports the bindings B.

$$\frac{}{E \vdash_{\mathsf{pat}} x^{?} \ : \ T \ \rightsquigarrow \ \{x : T\}}$$

$$\frac{E \vdash_{\mathsf{pat}} p_1 : T \rightsquigarrow B_1 \qquad E \vdash_{\mathsf{pat}} p_2 : T \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\mathsf{pat}} (p_1 \ \& \ p_2) \ : \ T \ \rightsquigarrow \ B_1 \uplus B_2}$$

$$\frac{E \vdash_{\mathsf{pat}} p_1 : T \rightsquigarrow B \qquad E \vdash_{\mathsf{pat}} p_2 : T \rightsquigarrow B}{E \vdash_{\mathsf{pat}} (p_1 \mid p_2) \ : \ T \ \rightsquigarrow \ B}$$

$$\frac{\begin{array}{c} E \vdash_{\mathsf{trm}} f : T \to (T_1, ..., T_n) \ \mathsf{option} \\ \forall i. \quad E \vdash_{\mathsf{pat}} p_i : T_i \rightsquigarrow B_i \qquad \forall i \neq j. \ B_i \# B_j \end{array}}{E \vdash_{\mathsf{pat}} f \ (p_1, ..., p_n) \ : \ T \ \rightsquigarrow \ \bigcup_i B_i}$$

$$\frac{E \vdash_{\mathsf{trm}} g : T \to \mathsf{bool}}{E \vdash_{\mathsf{pat}} g \ : \ T \ \rightsquigarrow \ \varnothing}$$

$$\frac{E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B_1 \qquad E, B_1 \vdash_{\mathsf{bbe}} b \rightsquigarrow B_2 \qquad B_1 \# B_2}{E \vdash_{\mathsf{pat}} (p \ \textbf{when} \ b) \ : \ T \ \rightsquigarrow \ B_1 \uplus B_2}$$

$$\frac{E \vdash_{\mathsf{pat}} p : T \rightsquigarrow B \qquad V \subseteq \mathsf{dom}\, B}{E \vdash_{\mathsf{pat}} (\textbf{restrict}\, V \, p) \ : \ T \ \rightsquigarrow \ B_{|V}}$$

Evaluation judgments, rules for terms and BBEs

- $t \Downarrow_{\mathsf{trm}} v$
- $b \Downarrow_{\mathsf{bbe}} r$
- $v \triangleright p \Downarrow_{\mathsf{pat}} r$

- r := Mismatch | Match M
- M maps variables to values
- We hide mutable store

$$\frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch} \qquad \mathbf{switch}\ c_2 \mid \ldots \mid c_n \Downarrow_{\mathsf{trm}} v}{\mathbf{switch}\ (\mathbf{case}\ b_1\ \mathbf{then}\ t_1) \mid c_2 \mid \ldots \mid c_n \ \Downarrow_{\mathsf{trm}}\ v}$$

$$\frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_1 \qquad \mathsf{Subst}(M_1, t_1) \Downarrow_{\mathsf{trm}} v}{\mathbf{switch}\ (\mathbf{case}\ b_1\ \mathbf{then}\ t_1) \mid c_2 \mid \ldots \mid c_n \Downarrow_{\mathsf{trm}} v} \qquad \frac{t \Downarrow v \qquad v \triangleright p \Downarrow_{\mathsf{pat}} r}{t\ \mathbf{is}\ p\ \Downarrow_{\mathsf{bbe}}\ r}$$

$$\frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch}}{b_1\ \mathbf{and}\ b_2 \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch}} \qquad \frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_1 \qquad \mathsf{Subst}(M_1, b_2) \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch}}{b_1\ \mathbf{and}\ b_2 \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch}}$$

$$\frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_1 \qquad \mathsf{Subst}(M_1, b_2) \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_2 \qquad M_1 \# M_2}{b_1\ \mathbf{and}\ b_2 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ (M_1 \uplus M_2)}$$

$$\frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_1}{b_1\ \mathbf{or}\ b_2 \Downarrow_{\mathsf{bbe}} \mathsf{Match}\ M_1} \qquad \frac{b_1 \Downarrow_{\mathsf{bbe}} \mathsf{Mismatch} \qquad b_2 \Downarrow_{\mathsf{bbe}} r}{b_1\ \mathbf{or}\ b_2 \Downarrow_{\mathsf{bbe}} r} \qquad \frac{b \Downarrow_{\mathsf{bbe}} r}{\mathbf{restrict}\ V\ b\ \Downarrow_{\mathsf{bbe}}\ r_{|V}}$$

Evaluation rules for patterns

$v \triangleright p \Downarrow_{\mathsf{pat}} r$ tests whether v matches p, returns Mismatch or Match M.

$$\frac{}{v \triangleright x^{?} \Downarrow_{\mathsf{pat}} \text{Match } \{x \mapsto v\}}$$

$$\frac{v \triangleright p \Downarrow_{\mathsf{pat}} \text{Mismatch}}{v \triangleright p \textbf{ when } b \Downarrow_{\mathsf{pat}} \text{Mismatch}}$$

$$\frac{v \triangleright p \Downarrow_{\mathsf{pat}} \text{Match } M \quad \text{Subst}(M, b) \Downarrow_{\mathsf{bbe}} \text{Mismatch}}{v \triangleright p \textbf{ when } b \Downarrow_{\mathsf{pat}} \text{Mismatch}}$$

$$\frac{v \triangleright p \Downarrow_{\mathsf{pat}} \text{Match } M_1 \quad \text{Subst}(M_1, b) \Downarrow_{\mathsf{bbe}} \text{Match } M_2 \qquad M_1 \# M_2}{v \triangleright p \textbf{ when } b \Downarrow_{\mathsf{pat}} \text{Match } (M_1 \uplus M_2)}$$

$$\frac{f(v) \Downarrow_{\mathsf{trm}} \text{None}}{v \triangleright f(p_1, ..., p_n) \Downarrow_{\mathsf{pat}} \text{Mismatch}}$$

$$\frac{f(v) \Downarrow_{\mathsf{trm}} \text{Some}(v_1, ..., v_n) \qquad \exists i.\ v_i \triangleright p_i \Downarrow_{\mathsf{pat}} \text{Mismatch}}{v \triangleright f(p_1, ..., p_n) \Downarrow_{\mathsf{pat}} \text{Mismatch}}$$

$$\frac{f(v) \Downarrow_{\mathsf{trm}} \text{Some}(v_1, ..., v_n) \qquad \forall i.\ \ v_i \triangleright p_i \Downarrow_{\mathsf{pat}} \text{Match } M_i \qquad \forall i \neq j.\ M_i \# M_j}{v \triangleright f(p_1, ..., p_n) \Downarrow_{\mathsf{pat}} \text{Match } (\bigcup_i M_i)}$$

$$\frac{g(v) \Downarrow_{\mathsf{trm}} \text{false}}{v \triangleright g \Downarrow_{\mathsf{pat}} \text{Mismatch}}$$

$$\frac{g(v) \Downarrow_{\mathsf{trm}} \text{true}}{v \triangleright g \Downarrow_{\mathsf{pat}} \text{Match } \varnothing}$$

$$\frac{v \triangleright p \Downarrow_{\mathsf{pat}} r}{v \triangleright \textbf{restrict } V\ p \Downarrow_{\mathsf{pat}} r_{|V}}$$

Type soundness theorem

Theorem (Preservation)

1. $(t \Downarrow_{\mathit{trm}} v) \ \wedge \ (\vdash_{\mathit{trm}} t : T) \ \Rightarrow \ (\vdash_{\mathit{trm}} v : T)$
2. $(b \Downarrow_{\mathit{bbe}} \mathit{Match}\ M) \ \wedge \ (\vdash_{\mathit{bbe}} b \rightsquigarrow B) \ \Rightarrow \ (\vdash_{\mathit{map}} M : B)$
3. $(v \triangleright p \Downarrow_{\mathit{pat}} \mathit{Match}\ M) \ \wedge \ (\vdash_{\mathit{pat}} p : T \rightsquigarrow B) \ \wedge \ (\vdash_{\mathit{trm}} v : T)$
   $\Rightarrow \ (\vdash_{\mathit{map}} M : B)$

where $\vdash_{\mathsf{map}} M : B$ is defined as:

$$\mathsf{dom}\, M = \mathsf{dom}\, B \ \wedge \ \forall\, x \in \mathsf{dom}\, M.\ \vdash_{\mathsf{trm}} M(x) : B(x)$$

Future work: prove *progress*, unless exhausting the branches of a switch.

Future work: Rocq formalization.

Naive compilation scheme

Translation into a core λ-calculus

- $[\![t]\!]$ translate a term.
- $[\![b]\!]^{u}_{u'}$ translate a BBE with two continuations: u for success and u' for failure. The term u may refer to variables exported by b.
- $\mathtt{y} \blacktriangleright [\![p]\!]^{u}_{u'}$ is the compilation of the code testing a value $\mathtt{y}$ against a pattern p, again with success and failure continuations u and u'.

Key definitions:

$$[\![\textbf{switch } (\textbf{case } b \textbf{ then } t) \mid c_2 \mid ... \mid c_n]\!] \equiv [\![b]\!]^{[\![t]\!]}_{[\![\textbf{switch } c_2 \mid ... \mid c_n]\!]}$$

$$[\![t \textbf{ is } p]\!]^{u}_{u'} \equiv \mathtt{let\ y} = [\![t]\!] \mathtt{\ in\ y} \blacktriangleright [\![p]\!]^{u}_{u'}$$

$$\begin{aligned}\mathtt{y} \blacktriangleright [\![f\ (p_1, ..., p_n)]\!]^{u}_{u'} \equiv\ & \mathtt{match}\ [\![\mathtt{f}]\!]\ \mathtt{y\ with} \\ & \mid \mathtt{Some\ (x1, ..., xn)} \rightarrow \\ & \qquad [\![(x_1 \textbf{ is } p_1) \textbf{ and } ... \textbf{ and } (x_n \textbf{ is } p_n)]\!]^{u}_{u'} \\ & \mid \_ \rightarrow u'\end{aligned}$$

$$\mathtt{y} \blacktriangleright [\![p \textbf{ when } b]\!]^{u}_{u'} \equiv \mathtt{y} \blacktriangleright [\![p]\!]^{[\![b]\!]^{u}_{u'}}_{\mathtt{u'}}$$

Correctness of the translation

Current formalization simplified to deterministic, terminating programs.

Theorem (Compilation perserves the semantics)

1. $(t \Downarrow_{\mathit{trm}} v) \ \wedge \ (\mathit{fv}(t) = \varnothing)$
   $\Rightarrow \ \ [\![t]\!] \Downarrow_{\mathit{ml}} [\![v]\!]$
2. $(b \Downarrow_{\mathit{bbe}} r) \wedge (\mathit{fv}(b) = \varnothing) \wedge \mathit{tr\text{-}cont}(r, u, u', w) \wedge \mathit{fv\text{-}cont}(r, u, u')$
   $\Rightarrow \ [\![b]\!]^{u}_{u'} \Downarrow_{\mathit{ml}} w$
3. $(v \triangleright p \Downarrow_{\mathit{pat}} r) \wedge (\mathit{fv}(p) = \varnothing) \wedge \mathit{tr\text{-}cont}(r, u, u', w) \wedge \mathit{fv\text{-}cont}(r, u, u')$
   $\Rightarrow \ ([\![v]\!] \blacktriangleright [\![p]\!]^{u}_{u'}) \Downarrow_{\mathit{ml}} w.$

where:

$$\mathsf{tr\text{-}cont}(r, u, u', w) \ := \ (\exists M.\, r = \mathsf{Match}\ M \ \wedge \ \mathsf{Subst}([\![M]\!], u) \Downarrow_{\mathsf{ml}} w) \ \vee \ (r = \mathsf{Mismatch} \ \wedge \ u' \Downarrow_{\mathsf{ml}} w)$$

$$\mathsf{fv\text{-}cont}(r, u, u') := (\mathsf{fv}(u') = \varnothing) \ \wedge \ (\forall\, M.\, r = \mathsf{Match}\ M \ \Rightarrow \ \mathsf{fv}(u) \subseteq \mathsf{dom}\, M)$$

Conclusion & Future work

Conclusion & Future work

Formalization in Rocq would be nice.

Implementation:

- to be integrated into the OptiTrust framework for source-to-source transformation; specifying which rewrite rules preserve the semantics
- perhaps also as a standalone pre-processor for OCaml?
- or maybe even as a conservative language extension?

Extensions:

- Interaction with exit-block construct, to support a backtracking switch construct (like e.g. in Prolog/LTac)
- Are all other practical programming patterns covered?