Partial Evaluation and Binding Time Analysis

Jurriaan Hage

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: jur@cs.uu.nl
Web pages: http://people.cs.uu.nl/jur/

May 25, 2011
Credits

- Stefan Holdermans - slides on partial evaluation
- Jurriaan Hage - slides on binding time analysis, based on Master Thesis of Guangyu Zhang.
Overview

- Partial Evaluation
- Binding Time Analysis
1. Partial Evaluation
Partial evaluation is concerned with specialising a program with respect to inputs that are known at compile time.

We distinguish between two forms of partial evaluation:

- **Online**: making decisions on what parts of a program to evaluate on the fly.
- **Offline**: making decisions on what parts of a program to evaluate based on annotations provided by a preprocessing phase (i.e., binding-time analysis).

We will only consider offline partial evaluation.
Why do partial evaluation?

▶ A general implementation is typically
  ▶ easier to write, shorter
  ▶ more generally usable
  ▶ slower

▶ Partial evaluation aims to compensate for the slowness.
▶ PE is an art: some expressions are better left unevaluated (at compile time).
  ▶ \textit{head} \[1..10000000\] versus \[1..10000000\]

▶ Does PE terminate?
The Knuth-Morris-Pratt linear string matcher can be obtained from the straightforward quadratic string matcher by specialisation for a given search string.

The Futamura projections

- specializing a language interpreter for a piece of source code yields an executable: \( \text{target} = \text{mix int source} \)
- specializing \( \text{mix} \) for an interpreter gives a compiler
- specializing again yields a compiler-generator: give it an interpreter, it will construct a compiler.
Partial evaluation: examples

\[
\begin{align*}
&\% : \text{pe} (2 + 3) \\
&\quad 5 \\
&\% : \text{pe} (\lambda x. 2 + 3 + x) \\
&\quad \lambda x. 5 + x \\
&\% : \text{pe} ((\lambda x. x + 3) 2) \\
&\quad 5 \\
&\% : \text{pe} (\lambda z. (\lambda f. \lambda x. \lambda y. f x y) (\lambda x. \lambda y. x + y) 2 z) \\
&\quad \lambda z. 2 + z
\end{align*}
\]
Black-box view

- Term $t$
- Binding-time analyser
  - Annotated term $\hat{t}$
  - Partial evaluator
  - Residual term $t'$
A partial evaluator performs three principle operations:

- **Evaluation**: reducing annotated terms to annotated values.
- **Residualisation**: transforming annotated terms into residual terms.
- **Lifting**: transforming annotated values into residual terms.
White-box view

annotated term $\hat{t}$ → \textit{residualise} → residual term $t$

$\text{lift}$

$\text{eval}$ → annotated value $\hat{v}$
We implement evaluation, residualisation, and lifting by means of metaprograms \textit{eval}, \textit{residualise}, and \textit{lift}:

\[
\begin{align*}
\text{eval} & : \widehat{Tm} \rightarrow \widehat{Val} \\
\text{residualise} & : \widehat{Tm} \rightarrow Tm \\
\text{lift} & : \widehat{Val} \rightarrow Tm
\end{align*}
\]

\textit{eval} and \textit{residualise} are partial functions with mutually disjoint domains.

\textit{lift} is a partial function over annotated values.
Static and dynamic terms

An annotated term is called static, iff it can be evaluated.

\[ \hat{t} \in \text{dom}(\text{eval}) \iff \hat{t} \text{ is static} \]

An annotated term is called dynamic, iff it can be residualised.

\[ \hat{t} \in \text{dom}(\text{residualise}) \iff \hat{t} \text{ is dynamic} \]

\[ \text{A term cannot be both static and dynamic.} \]
Well-formed and ill-formed terms

An annotated term is considered **well-formed**, iff it is static or dynamic.

\[(\hat{t} \text{ is static } \lor \hat{t} \text{ is dynamic}) \iff \hat{t} \text{ is well-formed}\]

An annotated term is considered **ill-formed**, iff it is not well-formed.

\[\neg (\hat{t} \text{ is well-formed}) \iff \hat{t} \text{ is ill-formed}\]

A well-formed term is either static or dynamic.
Taxonomy of annotated terms

annotated term

well-formed

static

ill-formed

dynamic
Typically, the source and target language of the partial evaluator (i.e., the language of annotated terms and the language of residual terms) have similar structures.

However, from the point of view of the partial evaluator, it is often more intuitive to simply identify residual terms with their textual representation:

\[ t \in Tm = \text{String} \]

So, residualise and lift behave as pretty printers that map dynamic annotated terms and annotated values, respectively, to strings.
What does Binding Time Analysis do?

- It attempts to discover which parts of the program may be evaluated at compile-time (S), and which must be evaluated at run-time (D).
- Maximize number of S-annotations
- PE may ignore certain S-annotations
- Work on (higher-order) Binding Time Analysis:
  - Dussart, Henglein, Mossin (1995): polyvariant, monomorphic, subtyping
  - Glynn, Stuckey, Sulzmann, Sondergaard (2001): polyvariant, polymorphic, subtyping
  - Zhang, Hage, Holdermans (2008): polyvariant, polymorphic, subeffecting
- Zhang built a number of variants to investigate differences in expressivity.
The term language

\[ n \in \text{Num} \quad \text{numerals} \]
\[ x \in \text{Var} \quad \text{variables} \]
\[ b \in \text{Boolean} \quad \text{booleans} \]
\[ op \in \text{Op} \quad \text{operators} \]
\[ t \in \text{Tm} \quad \text{terms} \]
\[ p \in \text{Prog} \quad \text{programs} \]

\[
\begin{align*}
  b & ::= \text{True} \mid \text{False} \\
p & ::= t \\
t & ::= n \mid b \mid x \mid \text{fun } x \Rightarrow t_1 \mid \text{fun rec } f \ x \Rightarrow t_1 \\
  & \mid t_1 \ t_2 \mid t_1 \ op \ t_2 \mid \text{let val } x = t_1 \ \text{in} \ t_2 \ \text{end} \\
  & \mid \text{let dyn val } x :: \ t_3 \ \text{in} \ t_1 \ \text{end} \mid \text{if } t_1 \ \text{then} \ t_2 \ \text{else} \ t_3
\end{align*}
\]
Remarks on the term language

- **fun** is for non-recursive, **fun rec** is for recursive function definitions.
- The **dyn val** construct is to be able to introduce dynamic values explicitly.
- Why do we need these?
- In a pure language (no side effects) every closed expression can be evaluated at compile-time.
- What about this \( tp \) component?

\[
\begin{align*}
\text{tp} & \in \text{Type} \quad \text{(mono)types} \\
\text{tp} & ::= \text{Nat} \mid \text{Bool} \mid \text{tp} \rightarrow \text{tp}
\end{align*}
\]
2. BTA-MMX - type system
What is BTA-MMX?

- The first and weakest in a series.
- BTA = Binding Time Analysis
- 1st M = monovariant
- 2nd M = monomorphic
- X = no weakening (no subeffecting, no subtyping)
Lattice of annotations - BTA-MMX

\( \varphi \in \text{Ann} \) annotations

\( \varphi ::= S \mid D \)

- \( S \) is bottom, \( \bot \), read as “known to be static”
- \( D \) is top, \( \top \), read as “assumed to be dynamic”
- The ordering is \( S \sqsubseteq D \).
Compared to Usage Analysis, we associate annotations directly with the types:

\[ \hat{\tau} \in \text{BTType} \quad \text{annotated types} \]

\[ \hat{\tau} ::= \ Nat^\varphi \mid \ Bool^\varphi \mid \hat{\tau}_1 \overset{\varphi}{\to} \hat{\tau}_2 \]

To extract the annotation of a type, we use \(|·|\):

- \(|Nat^\varphi| = \varphi\)
- \(|Bool^\varphi| = \varphi\)
- \(|\hat{\tau}_1 \overset{\varphi}{\to} \hat{\tau}_2| = \varphi\)
Not every annotated type makes sense: restrict our attention to those that are well-formed.

What does the function type $Nat^S \rightarrow D \rightarrow Nat^S$ mean?

- the function is unknown,
- but the argument and result are statically known.

Predicate $wft(\hat{\tau})$ is true if and only if $\hat{\tau}$ is well-formed:

- $wft(Nat^\varphi) = true$
- $wft(Bool^\varphi) = true$
- $wft(\hat{\tau}_1 \rightarrow \hat{\tau}_2) = wft(\hat{\tau}_1) \land wft(\hat{\tau}_1) \land \varphi \subseteq |\hat{\tau}_1| \land \varphi \subseteq |\hat{\tau}_2|$ 

Consequence: below a $D$ there are only $Ds$. 
Judgement form

Judgement has the usual form:

\[ \widehat{\Gamma} \vdash_{\text{BTA}} t : \widehat{\tau} \]

Read as “under annotated type environment \( \widehat{\Gamma} \), the term \( t \) has binding time annotated type \( \widehat{\tau} \)”.

Here we have the usual annotated type environments:

\[ \widehat{\Gamma} \in \text{TyEnv} \]

annotated type environments

\[ \widehat{\Gamma} ::= [] \mid \widehat{\Gamma}_1[ x \mapsto \widehat{\tau}] \]
BTA-MMX - literals and variables

\[ \hat{\Gamma} \vdash_{\text{BTA}} n : \text{Nat}^\varphi \]

\[ \hat{\Gamma} \vdash_{\text{BTA}} b : \text{Bool}^\varphi \]

Note: infinite family of rules, one for each \( n, b \) and \( \varphi \).

\[ \hat{\Gamma}(x) = \hat{\tau} \]

\[ \hat{\Gamma} \vdash_{\text{BTA}} x : \hat{\tau} \]
BTA-MMX - functions

Non-recursive:

\[ \hat{\Gamma} [ x \mapsto \hat{\tau}_x ] \vdash_{\text{BTA}} t_0 : \hat{\tau}_0 \quad \varphi \sqsubseteq |\hat{\tau}_0| \quad \varphi \sqsubseteq |\hat{\tau}_x| \]

\[ \hat{\Gamma} \vdash_{\text{BTA}} \text{fun } x \Rightarrow t_0 : \hat{\tau}_x \rightarrow \hat{\tau}_0 \]

Recursive:

\[ \hat{\Gamma} [ f \mapsto \hat{\tau}_x \quad \varphi \rightarrow \hat{\tau}_0 ] \vdash_{\text{BTA}} t_0 : \hat{\tau}_0 \quad \varphi \sqsubseteq |\hat{\tau}_0| \quad \varphi \sqsubseteq |\hat{\tau}_x| \]

\[ \hat{\Gamma} \vdash_{\text{BTA}} \text{fun rec } f \ x \Rightarrow t_0 : \hat{\tau}_x \rightarrow \hat{\tau}_0 \]

Note the built-in well-formedness check.
BTA-MMX - applications

Note the built-in well-formedness check.

We assume \( \text{op} \) has type \( \tau_1 \rightarrow \tau_2 \rightarrow \tau \).
Here, \( dyn(tp) \) constructs an annotated type decorated withDs only.
\[
\hat{\Gamma} \vdash_{\text{BTA}} t_1 : \text{Bool} \quad \hat{\Gamma} \vdash_{\text{BTA}} t_2 : \hat{\tau} \quad \hat{\Gamma} \vdash_{\text{BTA}} t_3 : \hat{\tau} \\
\hat{\Gamma} \vdash_{\text{BTA}} \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \hat{\tau}
\]

- Essential ingredient: \( \varphi \sqsubseteq |\hat{\tau}| \).
- Without it, a static then and else part could lead to a static conditional.
- Even if the condition is known to be dynamic.
BTA-MMX - the program

\[ \hat{\Gamma} \vdash_{\mathrm{BTA}} t : \hat{\tau} \quad D \sqsubseteq |\hat{\tau}| \]

\[ \hat{\Gamma} \vdash^p_{\mathrm{BTA}} t : \hat{\tau} \]

- Programs are expressions, but they must always be \( D \) at top-level.
- Reason: we must generate code for the program, even if the expression it contains evaluates to a static value.
- It also triggers setting annotations to \( D \).
3. BTA-MMX - algorithm
An algorithm for BTA-MMX

- A variant op algorithm \( \mathcal{W} \).
- During an AST-traversal we compute
  - the types of variables by means of unification
  - constraints on the binding-time annotations
- We give a separate solver to compute the solution for the constraints.
Augmented types

- We compute only monotypes, but do need type variables during algorithm execution.
- When we encounter a variable, we set its type to $\alpha$ to indicate it is unconstrained.
- Augmented type = old types + variables.

\[
\begin{align*}
\hat{\tau} & \in \text{BTTType} & \text{augmented types} \\
\varphi & \in \text{Ann} & \text{augmented annotations} \\
\alpha & \in \text{TyVar} & \text{type variables} \\
\beta & \in \text{AnnVar} & \text{annotation variables}
\end{align*}
\]

\[
\begin{align*}
\hat{\tau} & ::= \text{Nat}^{\varphi} | \text{Bool}^{\varphi} | \hat{\tau}_1 \varphi \hat{\tau}_2 | \alpha \\
\varphi & ::= S | D | \beta
\end{align*}
\]
As usual, during \( \mathcal{W} \) we compute (part of) the solution in the form of a substitution.

In this case, the substitution maps type variables to augmented types and maps annotation variables to augmented annotations.

In contrast with the thesis, we use a single substitution:

\[
\theta :: (\text{TyVar} \cup \text{AnnVar}) \rightarrow (\hat{\text{BTType}} \cup \hat{\text{Ann}})
\]

We assume substitutions are well-formed.

Substitutions are applied recursively (to augmented types) in the usual way.
Unification

- Computes the substitution that will make two augmented types the same (if it exists).
- If $\theta = \mathcal{U}_{\text{BTA}}(\hat{\tau}_1, \hat{\tau}_2)$, then $\theta \hat{\tau}_1 = \theta \hat{\tau}_2$.
- We need to perform unification on annotations and types.
- For simplicity, I combine these together.
- Cases for $S$ and $D$ are not needed: $\mathcal{W}_{\text{BTA}}$ only puts annotation variables into types.
- During unification, we only need to identify annotation variables.
The unification procedure

- $\mathcal{U}_{BTA}(Nat^{\beta_1}, Nat^{\beta_2}) = [\beta_1 \mapsto \beta_2]$
- $\mathcal{U}_{BTA}(Bool^{\beta_1}, Bool^{\beta_2}) = [\beta_1 \mapsto \beta_2]$
- $\mathcal{U}_{BTA}(\alpha, \alpha) = []$
- $\mathcal{U}_{BTA}(\alpha, \hat{\tau}) = [\alpha \mapsto \hat{\tau}]$ if $\alpha$ does not occur in $\hat{\tau}$
- $\mathcal{U}_{BTA}(\hat{\tau}, \alpha) = [\alpha \mapsto \hat{\tau}]$ if $\alpha$ does not occur in $\hat{\tau}$
- $\mathcal{U}_{BTA}(\hat{\tau}_1 \xrightarrow{\beta_1} \hat{\tau}_2, \hat{\tau}_3 \xrightarrow{\beta_2} \hat{\tau}_4) =$
  
  let $\theta_0 = [\beta_1 \mapsto \beta_2]$
  $\theta_1 = \mathcal{U}_{BTA}(\theta_0 \hat{\tau}_1, \theta_0 \hat{\tau}_3)$
  $\theta_2 = \mathcal{U}_{BTA}(\theta_1 (\theta_0 \hat{\tau}_2), \theta_1 (\theta_0 \hat{\tau}_4))$

  in $\theta_2 \circ \theta_1 \circ \theta_0$

- Substitutions are immediately incorporated: no need to unify substitutions.
As read from the type rules, constraints have just a single form, $\varphi \sqsubseteq \hat{\tau}$.

In the constraint language, we write $\varphi \leq_{h} \hat{\tau}$.

Solving constraints leads to substitutions: $[\beta \mapsto D]$ solves $D \leq_{h} Nat^\beta$.

What about $D \leq_{h} \alpha$?

Shape of $\alpha$ is undetermined, so no sensible answer yet.

Therefore, some constraints may be left unsolved.
Constraint definitions

\[ c \in \text{Con} \quad \text{constraints} \]
\[ cs \in \text{ConSet} \quad \text{constraint sets} \]

\[
c ::= \varphi \leq_h \hat{\tau}
\]
\[
 cs ::= [] \mid c : cs
\]
Constraint solver

- \textit{solve} takes a constraint set \( cs \), and return a triple \((cs', cs'', \theta)\), where
  - \( cs' \) the constraints that were solved
  - \( cs'' \) the constraints that could not be solved yet
  - \( \theta \) the substitution that solves the constraints in \( cs' \)

- The solver in the thesis is rather complicated and ad-hoc.
- Currently, I would use a worklist algorithm that
  - starts
4. Polymorphism and Polyvariance
5. Subeffecting or Subtyping