Partial Evaluation with Partially Static Operations

Peter Thiemann
University of Freiburg, Germany
thiemann@acm.org

Abstract
Partial evaluation distinguishes between different binding times when manipulating values in a program. A partial evaluator performs evaluation steps on values with a static binding time whereas it generates code for values with a dynamic binding time. Binding-time descriptions have evolved from monolithic to fine-grained, partially static data structures where different components may have different binding times.

We consider the next step in this evolution where not just data constructors but also operations can give rise to partially static structures, which are processed by the partial evaluator. We introduce this notion in an online setting, generalize it, generate a binding-time analysis for it, and specify an offline program generator supporting partially static operations with it. We prove the correctness of the binding-time analysis as well as the correctness and the completeness of the offline specialization with respect to the online specialization.

General Terms Experimentation, Languages, Theory
Keywords partial evaluation, algebraic simplification

1. Introduction
A program specialization transforms a program by taking advantage of known properties of the input data. The goal is to improve on some non-functional aspect of the program, such as efficiency, while keeping the semantics unchanged. Partial evaluation restricts the range of the properties considered by the specializer to being equal to a particular value. The degree of knowledge about a value is called its binding time.

In their seminal work on self-applicable partial evaluation, Neil Jones, Peter Sestoft, and Harald Søndergaard [6] devise a program analysis that assigns one of two monolithic binding times to each subexpression of a program: “Static” means that a value is fully known at specialization time, whereas “dynamic” classifies a value that exposes enough of its structure to enable additional static analysis that assigns one of two monolithic binding times to each subexpression of a program: “Static” means that a value is fully known at specialization time, whereas “dynamic” classifies a value that exposes enough of its structure to enable additional static analysis. Subsequent work has strengthened the power of the specializer to being equal to a particular value. The degree of knowledge about a value is called its binding time.

Our approach generalizes the idea of partially static data in the quest to improve the specialization of numeric code. As an example, let a, b, and d be static numbers, let c be a dynamic number, and apply a partial evaluator to the expression

$$ (a + b) + (c + d). $$

A binding-time analysis annotates each program construct with a binding time, either S for static or D for dynamic. This annotation indicates the execution time of the expression. The analysis also inserts lifting operations \([\cdot]\) that transform a number from its run-time representation to its source representation. The resulting annotated expression reads as follows:

$$ [(a +^S b)] +^D (c +^D [d]). $$

This annotation instructs the partial evaluator to add a and b at specialization time. Next, it lifts the sum a + b and d to run time and generates code for the remaining two additions.

However, the annotation in (1) is overly conservative. If the specializer were aware of the commutativity and associativity of addition, then it could reorder the expression before specialization:

$$ ((a + b) + d) + c $$

This equivalent expression has better binding-time properties. Two additions can be executed at specialization time and just one addition is deferred to run time.

$$ [(a +^S b) +^S d] +^D c$$

This paper examines ways of incorporating laws like commutativity and associativity in a partial evaluation algorithm without loosing its compositionality. The main idea is to have the specializer manipulate a symbolic, partially static representation of each value that exposes enough of its structure to enable additional static operations and to decrease the number of dynamic operations.

Preliminaries and Notation
Let \( \text{Var} \) be an infinite set of variables and \( \Theta \) be a ranked alphabet of operator symbols. For \( n \geq 0 \), write \( \Theta^{(n)} \) for the set of \( n \)-ary symbols. Let \( f \) range over elements of \( \Theta \) and let \( T_{\Theta}(\text{Var}) \) be the set of \( \Theta \)-terms over the variables:

- \( \text{Var} \subseteq T_{\Theta}(\text{Var}) \);
- for all \( n \geq 0, f \in \Theta^{(n)} \) and \( t_1, \ldots, t_n \in T_{\Theta}(\text{Var}) \) implies \( f(t_1, \ldots, t_n) \in T_{\Theta}(\text{Var}) \).

The height \( h(t) \) of a term \( t \) is defined recursively as usual.

\[
\begin{align*}
  h(x \mid x \in \text{Var}) &= 0 \\
  h(f(t_1, \ldots, t_n)) &= 1 + \max\{h(t_i) \mid 1 \leq i \leq n\}
\end{align*}
\]

If \( R \subseteq T_{\Theta}(\text{Var}) \times T_{\Theta}(\text{Var}) \) is a set of left-linear equations over terms with \( l \rightarrow r \in R \) implies \( \text{var}(r) \subseteq \text{var}(l) \), then the rewrite relation induced by \( R, R \vdash t \rightarrow t' \), is defined as the least relation...
obeying the inference rules

\[
\begin{align*}
\text{RAPPLY} & \quad r \to t \in \mathcal{R} \quad \sigma : \text{Var} \to T_{\text{sort}}(\text{Var}) \\
\mathcal{R} & \vdash \sigma(t) \to \sigma(r) \\
\text{RContext} & \quad \mathcal{R} \vdash t_i \to t_i' \\
& \quad \mathcal{R} \vdash f(\ldots,t_i,\ldots) \to f(\ldots,t_i',\ldots)
\end{align*}
\]

In rule RAPPLY, \( \sigma \) is a substitution, which is extended to terms in the usual, homorphic way. When writing \( r \to t \in \mathcal{R} \), it is understood that this notation obtains a fresh variant of the rule such that the variables appearing in it are nowhere else used. The symbol \( \to \) denotes the symmetric closure of relation \( \to \) and \( \leftarrow \sigma \) denotes its reflexive, transitive, and symmetric closure.

The notation \( f : A \to B \) specifies a partial function from \( A \) to \( B \) with domain \( \text{dom}(f) \) and range \( \text{ran}(f) \).

From now on, let \( \Omega \) be the ranked alphabet of the usual arithmetic operations including all numeric literals \( \mathbb{F} \) as constant symbols. Let further \( \mathcal{R}_\Omega \) be the set equations defining the usual arithmetic laws for rings (or fields, as appropriate).

Outline

Section 2 introduces specialization with partially static operations (PSO) with examples and formalizes the specialization of primitive operations. Section 3 extends this core specialization to a full online specializer with PSO and states a staging theorem. Section 4 develops a pattern-based binding-time analysis as an abstraction of the online specializer. Section 5 reconsiders the specialization of primitive operations by paying closer attention to staging. Section 6 develops the offline program generator from the results of the binding-time analysis. Section 7 reports on the status of the project and the implementation. Finally, Section 8 discusses related work and Section 9 concludes.

2. Specialization with Partially Static Operations

The specialization algorithm for partially static operations represents values in the form of admissible patterns \( P \subseteq T_{\text{sort}}(\text{Var}) \), which is a set of terms of bounded height that includes at least all variables as well as first-order constants drawn from set \( \mathbb{F} \). The smallest set of admissible patterns is thus \( P_{\text{old}} := \mathcal{R} \cup \text{Var} \), which is contained in any other admissible pattern set. An operation \( f \in \mathcal{G}^{(n)} \) is extended to pattern arguments \( p_1, \ldots, p_n \in P \) by first applying the operation symbolically. The resulting term \( t := f(p_1, \ldots, p_n) \) is not necessarily a pattern, anymore: with pattern set \( P_{\text{old}} \) the term \( t \) is never a pattern! Hence, the specializer applies laws and evaluation rules to simplify \( t \) with the goal of transforming it into an equivalent term \( t' \) such that \( t' \) is a pattern, again.

This rule is sufficient to explain the simplification of arithmetic terms like the following (where underlining marks variables that stand for dynamic data):

- \( 17 + 4 \) simplifies to 21 by evaluation,
- \( x + 0 \) simplifies to \( x \) by the unit law, and
- \( x + 0 \) simplifies to 0 by annihilation.

However, the term \( x + 1 \) cannot be simplified by evaluation or by applying laws. There are two ways to proceed. If \( x + 1 \in P \) is a pattern, then this expression is returned as a value. Otherwise, the specializer needs to generate code for the term as in \( \text{let } z = x + 1 \text{ in } \overline{z} \), where the definition \( z = x + 1 \) signifies generated code and the variable \( z \) is the pattern that is further processed. This style of code generation explains why patterns must include dynamic variables.

| Variable Env | \( \Delta \) ::= \cdot | \Delta, x |
| Definitions | \( \text{Def} \ni d \quad ::= \varepsilon \mid \varepsilon = t \mid d ; d \)
| Configuration | \( (d,t) \in \text{Def} \times T(\mathbb{F}, \text{Var}) \)
| OK-TERM-CONST | \( \Delta \vdash c \text{ OK} \)
| OK-TERM-OP | \( \Delta, x \vdash x \text{ OK} \)
| OK-DEF | \( \Delta \vdash t \text{ OK} \)
| OK-SEQ | \( \Delta \vdash d_1 \text{ OK} \)
| OK-EQ | \( \Delta \vdash \Delta \rightarrow x \leftarrow \Delta, \overline{x} \)
| OK-CONF | \( \Delta \vdash \langle d, t \rangle \text{ OK} \)

\[ \text{Figure 1. Well-formed configurations.} \]

In general, the specializer decomposes the transformed term \( t' \) into a pattern \( p \) and a substitution \( z_i \mapsto t_i' \) such that specializing \( t' = p(z_1 \mapsto t_1, \ldots) \) generates definitions for all \( z_i = t_i' \) and returns the pattern \( p \). This style of generating auxiliary definitions (but without intermixed law applications and simplifications) is well known as continuation-based partial evaluation [10]. Law applications and subterm abstractions are chosen so as to minimize the number of operations in the generated code.

The simplifier is the component of the partial evaluator that decides what to transform into a pattern and what to generate code for. It transforms a term into an equivalent output pattern along with zero or more definitions, which have been abstracted to make the resulting term fit into a pattern. The definitions correspond to generated code whereas the pattern is the structure that is further propagated by the specializer. The generated code may bind variables used in the output pattern.

Figure 1 defines the entities used in the specification of the simplifier. A configuration of the simplifier is a pair of a list of definitions and a transformed term, which is considered as a rewrite relation. All rewriting steps apply in an arbitrary context \( \mathcal{K} \) in the term component. The definitions component gathers generated code.

Rule RAPPLY applies a rewriting rule from \( \mathcal{R} \) to the term in the usual way. It replaces a substitution instance of a rule by its appropriately instantiated right side. Rule EVAL performs an evaluation step. It replaces an application of an operation \( f \) to all constants by its value. Here, \( [f] \) is the interpretation of the operation \( f \), such that \((f) \) applied to constants is again a constant in \( \mathcal{K} \). Rule FREEZE freezes a constant by abstracting it to a dynamic variable. Rule GEN abstracts an application of an
The rules from \( R \) apply rewriting to it. Lemma 1. to a list of definitions and a dynamic variable. The normal goal, Minimality: Inspection of the rules shows that rewriting preserves well-formation. \( S\mathcal{L}_O : T\Theta(\text{Var}) \rightarrow \text{Def} \times O \) \( S\mathcal{L}_O(s) = (d,t) \) where \( t \in O \) such that Correctness: \( R \vdash ([], s) \Rightarrow^* (d,t) \) Minimality: for all \( d', t' \) such that \( R \vdash ([], s) \Rightarrow^* (d', t'), |d| \leq |d'| \)

![Figure 2. Concrete Rewriting.](image)

![Figure 3. Specification of the simplifier.](image)

operation to constants or dynamic variables to code by assigning the application to a fresh variable and replacing it with the variable. At least one dynamic variable argument is needed, otherwise the operation could be evaluated. To ensure that rewriting can always reach a pattern, the rules \( \text{FREEZE} \) and \( \text{GEN} \) together ensure that every term can be rewritten to a list of definitions and a dynamic variable. The normal goal, however, is to find a more interesting pattern by rewriting. Inspection of the rules shows that rewriting preserves well-formation.

Lemma 1. If \( \Delta \vdash (d,t) \text{ OK and } R \vdash (d,t) \Rightarrow (d', t') \), then \( \Delta \vdash (d', t') \text{ OK.} \)

As an example, consider the term from the introduction \((a + b) + (c + d)\)

and apply rewriting to it.

![Figure 4. Pattern simplification and lifting the final result.](image)

\[
\begin{align*}
\text{Eval} & \quad ([], (a + b) + (c + d)) \\
\rightarrow & \quad ([], [a + b] + [c + d]) \\
\text{Rule} & \quad ([], [a + b] + (d + c)) \\
\rightarrow & \quad ([], ([a + b] + d) + c) \\
\text{Eval} & \quad ([], (a + b) + [d + c]) \\
\rightarrow & \quad ([], (a + b) + d) + [c, z] \\
\text{Gen} & \quad ([], ([a + b] + d) + [c, z]) \\
\rightarrow & \quad ([], [a + b] + [d + c], z) \\
\end{align*}
\]

Apparently, we obtain the desired result with the correct choice of rewriting steps. However, this choice is also a weakness. This rewriting relation is highly non-deterministic. Different rules from \( R \) may be applicable in several places in the term. The rules \( \text{Eval} \) and \( \text{Gen} \) may be applicable in several places. The \( \text{FREEZE} \) rule may be applied to an arbitrary constant in the term. Nevertheless, rewriting serves as the basis of the specification for the simplifier shown in Figure 3. The simplifier \( S\mathcal{L}_O \) is parameterized over a set \( O \) of admissible output terms, where \( \text{Var} \subseteq O \subseteq T\Theta(\text{Var}). \) It picks one particular pair of list of definitions and output term such that the original term rewrites to it and, among all such pairs, the cost of the list of definitions, indicated by \(|d|\), is minimal. One suitable cost measure is the length of the list which corresponds to the number of generated operations, but other measures may be used just as well. Freezing need not be considered in the rewriting for \( S\mathcal{L}_O \) if \( \text{Var} \subseteq O \). Figure 4 defines lifting \( \mathcal{L} \) as an instance of simplification. Lifting is needed in some places, for instance to produce the final result, where the output of the specializer must be code.

The specialization of an operation relies on simplification. It is specified by the function \( OP_n \), which applies a \( n \)-ary operator to \( n \) patterns and simplifies the result.

\[
OP : (n : N) \rightarrow (\Theta^n : \rightarrow P^n : \rightarrow \text{Def} \times P) \\
OP_n(x)(p_1, ..., p_n) = S\mathcal{L}(f(p_1, ..., p_n))
\]

The minimality criterion of \( SI \) rules out trivial solutions. 2.1 Example: Addition As an example, consider terms with the addition operator and the pattern set \( P_{\text{lin}} = P_{\text{add}} \cup \{ \text{Var} + d \mid x \in \text{Var}, d \in \mathbb{Z} \} \), where \( \text{Var} + d \) indicates a dynamic variable. For \( a, b, c, d \in \mathbb{Z} \) and \( \zeta \) a dynamic variable and writing \([a + b]\) for the number resulting from adding the numbers \( a \) and \( b \), the addition operator behaves as follows:

\[
\begin{align*}
(1) & \quad OP_2(+) (a, b) \neq ([\zeta = a + b], \zeta) & \text{not minimal} \\
(2) & \quad OP_2(+) (a, b) = ([\zeta = a + b]) & \text{ok} \\
(3) & \quad OP_2(+) (\zeta, d) = ([\zeta + d]) & \text{ok} \\
(4) & \quad OP_2(+) (a, \zeta + d) = ([\zeta + [a + d]]) & \text{ok}
\end{align*}
\]

The first line shows that an operation applied to constants must be evaluated as in the second line. Generating code in this situation would violate minimality. Line (3) assembles a pattern, which cannot be evaluated further. Line (4) associates commutativity to simplify to a pattern, still without generating code. These definitions are sufficient to process the introductory example of adding \((a + b) + (c + d)\). To visualize the progress of evaluation, we enclose each result pattern in angle brackets.

\[
\begin{align*}
(1) & \quad ((a + b)) + ((c + d)) \\
(2) & \quad ([a + b]) + ((c + d)) \\
(3) & \quad ([a + b]) + ([c + d]) \\
(4) & \quad ([a + b] + d) + ([c + d]) \\
\end{align*}
\]

2.2 Example: Exponentiation As another example, consider the specialization of integral exponentiation.

\[
power(x, n) = \\
\text{if } n = 0 \text{ then } 1 \text{ else } x * power(x, n-1)
\]

If \( x \) is dynamic and \( n \) is static, then the operations \(==\) and \(\rightarrow\) are always evaluated and do not lead to code generation. Standard binding-time analysis classifies both branches of the conditional as dynamic, so that the constant 1 becomes lifted and the operator
A top-level specialization and PE Env power (x) = let z1 = x*x in let z2 = x*z1 in z2

The simplification and the specialization of operators defined in the preceding section can form the basis of an online specializer. The specification unfolds all function calls, but it is easy to extend the terms with a conditional, a dynamic multiplication. The binding-time annotated * becomes a dynamic multiplication. The binding-time annotated version of power looks as follows.

\[
\text{power}_0 (x) = 1 \\
\text{power}_1 (x) = \text{let z} = x*1 \text{ in z} \\
\text{power}_2 (x) = \text{let z1} = x*1 \text{ in let z2} = x*z1 \text{ in z2}
\]

Of course, the code of power can be massaged to omit the multiplication by one, but it would be preferable if it was never generated in the first place.

This task turns out to be easy with partially static values using standard techniques\[5\]. In the second case, there is no choice: multi-

sions. Both return a list of definitions in the first component of their result, but while \( \mathcal{P}E \) returns a pattern for further PSO processing, \( \mathcal{P}E' \) composes \( \mathcal{P}E' \) with lifting and returns a (dynamic) variable. The specification unfolds all function calls, but it is easy to extend it with program point specialization to generate specialized functions using standard techniques\[5\].

The specialization works on the term language extended with conditionals, let expressions, and function calls. Its target language is the same as the source language when reading \( \text{let } d \text{ in } t \) as a nested let expression. Generally, the specialization is performed with respect to an environment \( \rho \in \text{Env} \). This environment maps each variable to a standard pattern, that is, either a first-order value or a dynamic variable.

The specialization of a constant yields the constant, which is always a pattern. A variable is looked up in the environment, which also results in a pattern. An (arithmetic) operation is specialized using the simplification function as specified in Section 2. Function calls are unfolded.

The specialization of the conditional first checks if the pattern is statically equivalent to true or false using the functions true? and false?, where true is encoded as any non-zero number and false as zero. If either of these tests is successful, the specializer drops the conditional and processes just the selected branch. Otherwise,
it specializes the branches as top-level expressions and generates code for a conditional. Code generation is done according to the conservative (read: non-duplicating) version of continuation-based partial evaluation [10] where context propagation stops at conditionals. That is, the branches of the conditional are specialized to top-level code using $\mathcal{P}\mathcal{E}$ and the local definitions for each branch are output in the respective branch of the generated conditional.

The correctness of the specializer is stated in the following conjecture. It basically specifies a staging property: Evaluating a program specialized to an environment given by $\rho_1$ and then specializing the result with respect to $\rho_2$, yields a result equivalent to specializing the same program with respect to $\rho_2 \circ \rho_1$.

**Conjecture 1.** Suppose that $\rho_1, \rho_2 : \text{Var} \rightarrow P_{\text{std}}$. Let $\mathcal{P}\mathcal{E}'[[t]]\rho_1 = (d_1, p_1), \mathcal{P}\mathcal{E}'[[t]]\rho_2 = (d_2, p_2)$, and $\mathcal{P}\mathcal{E}'[[t]](\rho_2 \circ \rho_1) = (d_3, p_3)$. Then $p_2 = p_3$ and $d_3 \approx d_2$ (equal up to reordering of definitions).

The cognoscenti may wonder why the specializer is not written in monadic style. Indeed, the type constructor $\text{Env} \rightarrow \text{Def} \times \text{Def}$ is a reader monad combined with an output monad because lists with concatenation form a monoid. However, the treatment of the conditional requires a number special operators, which seem easier to read with all monad operations inlined, rather than having to look them up separately.

### 4. Binding-Time Analysis

Originally, binding-time analysis has been developed to achieve efficient self application [6]. Later it has become instrumental in creating powerful and efficient specialization algorithms (e.g., [4]) as well as in the direct derivation of code generators from binding-time annotated programs (the cogen approach) [9].

Using a binding-time analysis as a pre-pass to specialization proper yields an offline specializer. In such a specializer, the decisions whether to generate code or to evaluate an expression are taken by the binding-time analysis. The specializer itself just executes the annotations created by the analysis.

A binding-time analysis may be created by systematic abstract interpretation of an online specializer, like the one in Figure 6. There are two tasks in designing the analysis. First, to come up with a suitable abstract domain and, second, to define the transfer functions.

The idea for the analysis domain is to abstract over values that are static, but unknown at analysis time. To this end, we abstract the specialization-time values to abstract terms $\hat{T}$ defined by the grammar in Figure 7 along with the ordering $\subseteq \hat{T}$ that makes abstract terms into a lattice and the concretization function $\gamma$ that maps each abstract term to a set of terms, considered as an element in the power set lattice over terms. The alternatives for an abstract term are bottom $\bot$, a variable $X$, an element of the flat upper semilattice $\mathbb{T}^+$, a term with operator $f$, or a union of two abstract terms. The union operator is associative, commutative, and idempotent.

**Lemma 3.** $\gamma$ is monotone.

**Proof.** By induction on the proof of $\hat{t}_1 \subseteq \hat{t}_2$.

- $\bot \subseteq \hat{t}_i \subseteq \gamma(\hat{t}_i)$.
- $a \in T_f \vdash a \subseteq \hat{t}_i$.
- $\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_1 \cup \hat{t}_2$: obvious.
- $\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_2 \cup \hat{t}_1$: commutativity of $\cup$.

The corresponding abstraction function $\alpha : \mathbb{T}^{\text{std}} \rightarrow \hat{T}$ is determined as usual to make $(\alpha, \gamma)$ into a Galois connection.

$$\alpha(M) = \{ \hat{t} \in \hat{T} \mid M \subseteq \gamma(\hat{t}) \}$$

The thus constructed abstract domain is not yet suitable for analysis because it has infinite height. But as each specialization scenario comes with its own set $P$ of patterns, we define its associated analysis on the abstract pattern set $\hat{T}_P = \{ \alpha(P') \mid P' \subseteq P \}$, which is also a lattice under the ordering induced from $\hat{T}$. This lattice still has infinite height because we can form the union of an arbitrary number of constants. The solution is to consider union modulo a number of equations (beyond assuming commutativity, associativity, and idempotence of union):

$$\bot \cup \hat{t} = \hat{t} \quad \alpha \neq b \in \mathbb{T}^+ \quad a \cup b = \bot$$

$$\forall 1 \leq i \leq n \quad t_i \cup t'_i = t''_i$$

In the thus reduced domain, each element is either $\bot$ or a finite union of distinct alternatives from Figure 7.

Given this domain construction, the abstraction of the operations is entirely standard.

$$\overline{OP} : \{ n : \mathbb{N} \rightarrow \Omega(n) \rightarrow \hat{T}\hat{P} \rightarrow \hat{T}_P$$

$$\overline{OP}_n(f)(\hat{p}_1, \ldots, \hat{p}_n) =$$

$$\alpha(\{ \#(\overline{OP}_n(f)(p_1, \ldots, p_n)) \mid p_i \in \gamma(\hat{p}_i) \})$$

The function $\#P$ projects the second component of a pair. By definition, $\overline{OP}_n(f)$ is additive for each $f$.

For example, the abstract domain derived from the standard pattern set $P_{\text{std}}$ can be described by the following grammar for elements.

$$\hat{T}_{\text{std}} := \bot \mid X \mid \mathbb{T}^+ \mid \mathbb{T}^+ \cup X$$

<table>
<thead>
<tr>
<th>Abstract terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{T} := \bot \mid X \mid \mathbb{T}^+ \mid \mathbb{T}^+ \cup X$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot \subseteq \hat{t}$</td>
</tr>
<tr>
<td>$a \in T_f \vdash a \subseteq \hat{t}$</td>
</tr>
<tr>
<td>$\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_1 \cup \hat{t}_2$</td>
</tr>
<tr>
<td>$\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_2 \cup \hat{t}_1$</td>
</tr>
<tr>
<td>$\hat{t}_1 \cup \hat{t}_2 \subseteq \hat{t}_2 \cup \hat{t}_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concretization function $\gamma : \hat{T} \rightarrow 2^{\Omega(\hat{T})}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(\bot) = \emptyset$</td>
</tr>
<tr>
<td>$\gamma(\hat{t}_1 \cup \hat{t}_2) = \gamma(\hat{t}_1) \cup \gamma(\hat{t}_2)$</td>
</tr>
<tr>
<td>$\gamma(\alpha) = { a }$</td>
</tr>
<tr>
<td>$\gamma(\forall \hat{t}_1, \ldots, \hat{t}_n) = { f(t_1, \ldots, t_n) \mid t_i \in \gamma(\hat{t}_i) }$</td>
</tr>
</tbody>
</table>

Figure 7. Abstract pattern domain.
The only non-trivial case is the right-hand side for the conditional. As true?, false?, and dyn? are monotonic functions from $\top_P$ to the lattice \{false, true\}, each of the conditions involving these functions are also monotonic, and hence their join is monotonic, too.

\[
\begin{align*}
\mathcal{OP}_2(+) & (a, b) = [a + b] \\
\mathcal{OP}_2(+) & (\top_T, b) = \top_T \\
\mathcal{OP}_2(+) & (X, b) = X + b \\
\mathcal{OP}_2(+) & (X + a, b) = X + a + b \\
\mathcal{OP}_2(+) & (X + T_T, b) = X + T_T \\
\mathcal{OP}_2(+) & (\top_T, T_T) = \top_T \\
\mathcal{OP}_2(+) & (\top_T, X) = \top_T \\
\mathcal{OP}_2(+) & (X + a, X) = X + a \\
\mathcal{OP}_2(+) & (X + T_T, X) = X + T_T \\
\mathcal{OP}_2(+) & (X + a, X) = X + a + b \\
\mathcal{OP}_2(+) & (X + T_T, X) = X + T_T \\
\mathcal{OP}_2(+) & (X + T_T, X + T_T) = \top_T
\end{align*}
\]

\hspace{1cm} Figure 8. Abstract addition based on $P_{\text{tin}}$.

Using this domain and then abstracting the multiplication operation from the exponentiation example yields (incomplete):

\[
\begin{align*}
\mathcal{OP}_2(+) & (X, 1) = X \\
\mathcal{OP}_2(+) & (X, X) = X \\
\mathcal{OP}_2(+) & (X, 1 \cup X) = X \\
\end{align*}
\]

As another example, using $\hat{T}_{P_{\text{tin}}}$ and abstracting the addition operation as before yields the table in Figure 8. Such a table could be precomputed from the underlying pattern set using the definition of the abstract operations. Alternatively, the table could be computed on demand and cached during the analysis.

The binding-time analysis is straightforward to obtain from the online specializer with PSO as a function $\mathcal{B} : T_{\text{in}}(X, \text{Var}) \rightarrow \text{Env}$ where $\text{Env} = \text{Var} \mapsto \hat{T}_P$ is the set of abstract environments. Figure 9 contains the definition of the analysis, which is largely a transliteration. The most interesting part of the analysis is the definition of the conditional. It relies on a number of auxiliary functions, true?, false?, and dyn?, that check whether an abstract pattern may be statically true, false, or dynamic. These functions are also shown in Figure 9. They are constructed systematically from their counterparts in the online specializer. For example:

\[
\begin{align*}
\text{true?}(\hat{\rho}) &= \bigcup \{\text{true?}(\rho) \mid \rho \in \gamma(\hat{\rho})\}
\end{align*}
\]

The join operation on the right-hand side considers \{false, true\} as a two element lattice with false $\sqsubseteq$ true.

There is one caveat in comparison to the online specializer. Because functions may be recursive, the termination of the analysis function cannot be guaranteed. Thus, an implementation needs to rely on a caching mechanism to compute the fixpoint of the analysis in a finite amount of time.

To ensure that this fixpoint exists, all transfer functions (the interpretations of the various syntactic constructions) must be monotonic functions on a lattice of finite height (i.e., $\hat{T}_P$).

**Lemma 4.** $\mathcal{B}$ is a monotonic function.

**Proof.** The only non-trivial case is the right-hand side for the conditional. As true?, false?, and dyn? are monotonic functions from $\hat{T}_P$ to the lattice \{false, true\}, each of the conditions involving these functions are also monotonic, and hence their join is monotonic, too.
\[ \gamma(B[t_0\ldots t_n]) \text{ for } 1 \leq i \leq n, \text{ and we need to show that } p \in \gamma(B[f(t_1,\ldots,t_n)]) \text{ too.} \]

\[
\begin{align*}
\gamma(B[f(t_1,\ldots,t_n)]) &= \gamma(\mathcal{O}_{\mathcal{P}}(f)(B[t_1]\ldots,B[t_n])) \\
&= \gamma(\alpha(\# \mathcal{O}_{\mathcal{P}}(f)(p_1,\ldots,p_n)) | p_i \in \gamma(B[t_i])) \\
&= \bigcup \gamma(\alpha(\# \mathcal{O}_{\mathcal{P}}(f)(p_1,\ldots,p_n)) | p_i \in \gamma(B[t_i])) \\
&\subseteq \bigcup \gamma(\# \mathcal{O}_{\mathcal{P}}(f)(p_1,\ldots,p_n)) | p_i \in \gamma(B[t_i])) \\
&= \{ \# \mathcal{S}(f(p_1,\ldots,p_n)) \} \\
&= \{ \emptyset \}
\end{align*}
\]

• \( t = \text{let } x = t_1 \text{ in } t_2 ; \)

\[
\begin{align*}
\mathcal{P}^\gamma \left[ \text{let } x = t_1 \text{ in } t_2 \right] &= \mathcal{P}^\gamma(t_1,d_2,p_2) \\
\text{where } (d_1,p_1) &= \mathcal{P}^\gamma(t_1,d_2,p_2) \\
\text{false} \wedge (d_1,p_1) &= \mathcal{P}^\gamma(t_2,d_2,p_2) \\
\text{else} \wedge (d_1,p_1) &= \mathcal{P}^\gamma(t_2,d_2,p_2) \\
\text{let } x = t_1 \text{ in } t_2 \right] &= \mathcal{P}^\gamma(t_1,d_2,p_2) \\
\text{let } x = t_1 \text{ in } t_2 \right] &= \mathcal{P}^\gamma(t_2,d_2,p_2)
\end{align*}
\]

Let \( \tilde{p}_1 = B[t_1] \tilde{p} \). By induction, \( p_1 \in \gamma(\tilde{p}_1) \) so that \( p[x \mapsto p_1] \). Hence, by further induction, \( p_2 \in \gamma(B[t_2]p[x \mapsto p_1]) \). Let \( \tilde{p}_2 = B[t_2] \tilde{p} \). By induction, \( p_1 \in \gamma(\tilde{p}_2) \) so that \( p[x \mapsto p_1] \). Hence, by further induction, \( p_2 \in \gamma(B[t_2]p[x \mapsto p_1]) \). Let \( \tilde{p}_0 = B[t_0] \tilde{p} \). By induction, \( p_0 \in \gamma(\tilde{p}_0) \), \( p_1 \in \gamma(\tilde{p}_1) \), and \( p_2 \in \gamma(\tilde{p}_2) \).

Case \( p_0 = c \neq 0 \):

true?\(p_0\) is true and \( c \in \gamma(\tilde{p}_0) \), which is equivalent to \( \tilde{p}_0 = c \cup \tilde{p} \) or \( \tilde{p}_0 = T \cup \tilde{p} \). In both cases, true?\(p_0\) is true so that \( B[if \ t_0 \ t_1 \ t_2 \rho] \supseteq \tilde{p}_1 \), which means that \( \gamma(B[if \ t_0 \ t_1 \ t_2 \rho]) \supseteq \gamma(\tilde{p}_1) \supseteq p_1 \).

Case \( p_0 = 0 \):

true?\(p_0\) is false and false?\(p_0\) is true and \( 0 \in \gamma(\tilde{p}_0) \), which is equivalent to \( \tilde{p}_0 = \emptyset \cup \tilde{p} \) or \( \tilde{p}_0 = T \cup \tilde{p} \). In both cases, false?\(p_0\) is true so that \( B[if \ t_0 \ t_1 \ t_2 \rho] \supseteq \tilde{p}_2 \), which means that \( \gamma(B[if \ t_0 \ t_1 \ t_2 \rho]) \supseteq \gamma(\tilde{p}_2) \supseteq p_2 \).

Case \( p_0 = x \):

true?\(p_0\) is false and false?\(p_0\) is false and \( x \in \gamma(\tilde{p}_0) \), which is equivalent to \( \tilde{p}_0 = X \cup \tilde{p} \). In this case, dyn?\(\tilde{p}_0\) is true so that \( B[if \ t_0 \ t_1 \ t_2 \rho] \supseteq X \), which means that \( \gamma(B[if \ t_0 \ t_1 \ t_2 \rho]) \supseteq \gamma(X) \supseteq \{ x \} \).

Case \( p_0 = f(\ldots) \):

true?\(p_0\) is false and false?\(p_0\) is false and \( f(\ldots) \in \gamma(\tilde{p}_0) \), which is equivalent to \( \tilde{p}_0 = f(\ldots) \cup \tilde{p} \). In this case, dyn?\(\tilde{p}_0\) =

\begin{align*}
\begin{array}{c|c|c|c}
\text{Iteration} & 1 & 2 & 3 \\
\hline
\text{power}(x,n-1) & \bot & 1 & 1 \cup \Delta \\
\text{x + power}(x,n-1) & \bot & \chi & \chi \\
\text{power}(x,n) = \text{if...} & 1 & 1 \cup \chi & 1 \cup \chi \\
\end{array}
\end{align*}

\textbf{Figure 10.} Binding-time analysis for power

\textbf{Staged Terms} \( s \in S_{\Sigma} = T_{\Sigma} \text{ (SVar, DVar)} \)

\textbf{Staged Configuration} \( \langle d,d,s \rangle \in \text{Def} \times \text{Def} \times S_{\Sigma} \)

\textbf{OK-STAGED-CONF} \quad \vdash d_1 \leadsto \Sigma \\
\Sigma \subseteq \text{SVar} \quad \Delta \subseteq \text{DVar} \\
\Sigma, \Delta \vdash d_2 \leadsto \Delta' \quad \Delta' + s \text{ OK} \\
\Delta \vdash \langle d_1,d_2,s \rangle \text{ OK}

\textbf{Figure 11.} Well-formedness of staged configurations.

\textbf{ARULE} \quad l \rightarrow r \in \mathcal{R} \\
\mathcal{R} \vdash \langle d_0,d_1,K[\sigma(l)] \rangle \rightarrow \langle d_0,d_1,K[\sigma(r)] \rangle

\textbf{AEVAL} \quad s = f(s_1,\ldots,s_n) \quad (\forall i, s_i \in \mathcal{F} \cup \text{SVar}) \\
\mathcal{R} \vdash \langle d_0,d_1,K[s] \rangle \rightarrow \langle d_0,|z = s|,d_1,K[z] \rangle

\textbf{AFREEZE} \quad a \in \mathcal{F} \cup \text{SVar} \quad z \in \text{DVar fresh} \\
\mathcal{R} \vdash \langle d_0,d_1,K[a] \rangle \rightarrow \langle d_0,d_1,|z = [a]|,K[z] \rangle

\textbf{AGEN} \quad s = f(z_1,\ldots,z_n) \quad (\forall i, z_i \in \text{DVar fresh} \\
\mathcal{R} \vdash \langle d_0,d_1,K[z] \rangle \rightarrow \langle d_0,|z = [z]|,K[z] \rangle

\textbf{Figure 12.} Staged Rewriting.

\begin{align*}
\text{true so that } B[if \ t_0 \ t_1 \ t_2 \rho] \supseteq X \text{ which means that } \\
\gamma(B[if \ t_0 \ t_1 \ t_2 \rho]) \supseteq \gamma(X) \supseteq \{ z \}.
\end{align*}

In the presence of recursion, the analysis requires the usual fix-point computation to obtain a safe result. Once a fixpoint is reached, each subexpression is annotated with a union of abstract patterns. These abstract patterns subsume the different patterns that may arise from this subexpression at specialization time. As patterns distinguish between static and dynamic contents, this analysis serves as a binding-time analysis for the specializer with partially static operations.

The table in Figure 10 shows an excerpt of the results of the corresponding fixpoint computation on the power example. The abstractions for 1, x, and n are, respectively, 1, X, and T_\mathcal{P}, which indicates that the analysis distinguishes between program constants 1 and static values T_\mathcal{P} that are unknown at analysis time as well as dynamic values X.

\section{5. Staged Operations}

There is one fly in the ointment when trying to use the information from a binding-time analysis. The definition of an abstract operation in equation (2) is a perfectly fine mathematical definition, but it is not suitable for effectively computing the abstract operation. Moreover, using the binding-time pattern for specialization
requires extra information about the simplifications performed during the analysis.

To obtain an effective definition that provides enough information requires another look at the rewrite system for simplification in Figure 2. The goal is to construct a version of this rewriting system that works on a slightly more concrete version of abstract terms and use that version to compute the abstract operations analogously to the computation of the concrete operations.

Instead of using abstract terms, the upcoming algorithm operates on terms with distinct notions of static and dynamic variables, SVar and DVar. The static variables are the “more concrete” version of the unknown number $\top$, they are bound to numbers which are known at specialization time, whereas dynamic variables concretize $\bar{X}$ and contain run-time numbers.

Figure 12 specifies the staged rewriting relation. Its configuration (defined in Figure 11) consists of two lists of definitions, one for static definitions and another for dynamic ones, and a staged output term. The static definitions collect static applications of operations to operands that are known at analysis time, but known at specialization time. The result of such a static operation is itself static, so it gets bound to a static variable from SVar. Variables ranging over code are now drawn from a set DVar of dynamic variables, marked by underlining as before. The well-formedness judgment $\Delta \vdash \langle d, d', s \rangle$ OK enforces that $s$ only refers to defined variables, that $d_1$ only defines static variables and does not refer to dynamic ones, and that definitions in $d_2$ may refer to all previously defined static and dynamic variables.

The rules ARULE, AFREEZE, and AGEN are transcriptions of the corresponding concrete rules, mutatis mutandis. The rule AEVAL requires some attention. An operation is statically applicable if all its arguments are known. In a pattern, there are three ways to express known subterms: a constant, an unknown constant represented by $\top$, and a previously computed value represented by a static variable. The rule checks that all arguments fall in one of these cases, creates a static definition for a fresh static variable, and replaces the operation by this variable.

Staged rewriting is correct if each staged rewrite sequence is mirrored by a concrete rewrite sequence. To this end, an approximation relation is defined between staged terms and definitions and their ordinary versions. In short, staged terms can abstract constants.

There exists a substitution of $\sigma$ with $\bar{\sigma}$ such that $t = \sigma(t)$ (using the obvious extension of $\vdash$ to substitutions). Clearly, there exist suitable $t' = \sigma(r)$ with $\bar{s}' \vdash \sigma(s)$, $d'_0 = d_0$, and $d'_1 = d_1$.

**Case AEVAL**: If $s = f(s_1, \ldots, s_n)$, then each $t$ with $s \vdash t$ has the form $t = (\bar{f}(b_1, \ldots, b_n)$ with all $b_i \in \mathbb{F}$. Hence, the EVAL rule replaces $t$ with $b = [\bar{f}(b_1, \ldots, b_n)$. The claim follows from $\bar{z} \geq b$, where $z \in SVar$ is the fresh variable, and because the dynamic definitions do not change.

**Case AFREEZE**: If $s = a \in \mathbb{F} \cup SVar$, then each approximation of $s$ is some $b \in \mathbb{F}$. Hence, rule FREEZE is applicable and yields a dynamic definition $\bar{z} = b$. The claim follows because the right-hand terms match and the dynamic definitions are extended in the same way.

**Case AGEN**: If $s = f(\bar{z}_1, \ldots, \bar{z}_n)$ where each $\bar{z}_i \in DVar$, then $t$ has the form $t = f(\bar{z}_1, \ldots, \bar{z}_n)$ where rule GEN is applicable and yields a new definition that matches the new dynamic definition. The right-hand terms also match.

Besides forming a basis for computing the abstract operations used in the binding-time analysis, the rewrite system is also a step towards the construction of a cogen. As the static data is not available when executing the cogen, the cogen also needs to generate code for the static operations. This code is gathered by the rewrite system in the static definitions. The dynamic definitions give rise to code that need to generate code.

The next question is whether the rewrite system is complete. This is an important question because it determines whether the offline generalizer based on the analysis has the same strength as the actual specialization from Figure 6.

Unfortunately, the answer to this question is no for the rewriting system from Figure 12. To see this, consider the term $(2 - 1) * \bar{z}$ where $\bar{z}$ is dynamic. The concrete rewriting system simplifies as follows:

$$\langle [], (2 - 1) * \bar{z} \rangle \Rightarrow \langle [], 1 * \bar{z} \rangle \Rightarrow \langle [], \bar{z} \rangle$$

But the staged system abstracts the result of $(2 - 1)$ to a static variable, which disables the application of the unit law for multiplication.

$$\langle [], (2 - 1) * \bar{z} \rangle \Rightarrow \langle [], (2 - 1) * \bar{z} \rangle$$

This shortcoming can be addressed by extending staged rewriting with a variant of AEVAL that works like EVAL. This rule, say, AEVAL1 replaces an operation $f(a_1, \ldots, a_n)$ with $a_i \in \mathbb{F}$ by $\langle [], a_1, \ldots, a_n \rangle$. It can be proven correct by a straightforward extension of Lemma 6.

But this extension is still not sufficient to obtain completeness. For completeness, it would be required that each concrete rewrite step has a corresponding staged rewrite step. However, consider $t = \bar{z} + 0$ and the staged term $s = \bar{z} + \bar{y}$. Clearly, $s \vdash t$, but while $s$ is irreducible in the staged system, a rewrite step is applicable in the concrete system

$$\mathcal{R} \vdash \langle [], \bar{z} + 0 \rangle \Rightarrow \langle [], \bar{z} \rangle$$

(3)

This deficiency can be remedied by extending rule ARULE to perform narrowing [16]. The basic idea of narrowing is to instantiate a term such that a rewriting step becomes applicable to the inside of the term. Here is a suitable replacement for rule ARULE:

**ARULE1**

$$s = f(\ldots), \quad l \rightarrow r \in \mathcal{R}, \quad \bar{s} \vdash \sigma \Rightarrow \bar{s} : \text{Var} \rightarrow S_2, \quad \bar{s}(\sigma) = \bar{s}(l) \quad \text{run}(\bar{s}(\text{Var}(s)) \subseteq \mathbb{F}, \quad \mathcal{R} \vdash \langle d_0, d_1, K[s] \rangle \rightarrow \bar{s}[\text{Var}(s)] \langle d_0, d_1, K[\bar{r}] \rangle$$

The first assumption states that the left-hand side of the rule should be rooted in a non-variable position of the term. The fourth assumption requires that $s$ and $l$ be unifiable. The last assumption restricts
\[ ST^2_0 \colon S_T \Rightarrow \text{Def} \times \text{Def} \times O \]
\[ ST^2_0(s) = (d_0, d_1, p) \text{ where } p \in O \text{ such that} \]

**Correctness:**
\[ R \vdash \{s\} \Rightarrow \Rightarrow \gamma \Rightarrow \langle d_0, d_1, p \rangle \]

**Minimality:**
\[ \text{for all } d_0, d_1, p, q \text{ such that } R \vdash \{s\} \Rightarrow \Rightarrow \gamma \Rightarrow \langle d_0, d_1, p \rangle, \]
\[ |d_0| \leq |d_1| \]

**Figure 13.** Specification of the staged simplifier.

The unification to only introduce constants in the subject term. Otherwise, an ARULE1-step might introduce new operations and hence cause non-termination. When performing the rewrite step, the instantiation of \( s \) that is needed to perform the rule application is a further output. Rewriting steps that do not instantiate the input term generate the identity substitution and the substitution associated to a sequence of rewriting steps is the composition of their individual substitutions.

To put this new rule to work, let \( s = x + y \) for \( y \in SVar \). Using rule ARULE1, there is a staged rewrite step mirroring the concrete step in (3).

\[ R \vdash x + y \Rightarrow \{y \mapsto 0\} \Rightarrow \]

These two extensions are sufficient to prove the staged system complete with respect to concrete rewriting.

**Lemma 7.** Suppose that \( R \vdash (d, t) \Rightarrow (d', t') \).

Then, for all \( s, d_0, d_1 \) such that \( (d_0, d_1, s) \Rightarrow (d, t) \), there exist \( d_0', d_1', s' \) and \( \sigma \) such that \( R \vdash (d_0, d_1, s) \Rightarrow \Rightarrow \sigma \Rightarrow (d_0', d_1', s) \) and \( (d_0, d_1, s) \Rightarrow (d, t) \).

**Proof.** Contexts are taken into account in the same way as in the proof of Lemma 6. Hence, w.r.o.g. the rest of this proof considers rewrite steps in the empty context.

**Case Rule:** In this case, \( t = f(t_1, \ldots, t_n) = \sigma(f) \) for some left-hand side \( l \) of a rule. Thus, any approximation of \( t \) has the form \( s = f(s_1, \ldots, s_n) \) with all occurrences of static variables distinct. To show that ARULE1 is applicable, it is sufficient to construct a unifying substitution \( \sigma \) by induction on \( l \).

Subcase \( l = f(l_1, \ldots, l_n) \). Thus, \( t = f(l_1, \ldots, l_n) \) and \( s = f(s_1, \ldots, s_n) \) and, by induction, there are unifying substitutions \( \sigma_1, \ldots, \sigma_n \) because the variable occurrences are distinct and because rules are left linear, set \( \sigma = \sigma_1 \circ \cdots \circ \sigma_n \).

Subcase \( l = a \in F \). Thus, \( t = a \) and either \( s = a \) and \( \sigma = id \) or \( s = x \in SVar \) and \( \sigma = [x \mapsto a] \).

Subcase \( l = x \in Var \). Thus, \( t \) and \( s \) are arbitrary such that \( s \geq t \) and \( \sigma = [x \mapsto s] \).

**Case Eval:** In this case, AEval or AEVAL1 are applicable.

**Case Freeze:** AFREEZE is applicable.

**Case Gen:** AGEN is applicable.

Just like in Section 3, the staged rewriting system can be used to define a staged simplification function and corresponding staged operations. Figure 13 contains the staged simplifier. It relies on the same ideas as the earlier simplification function. It returns two lists of definitions, corresponding to two stages of computation. The current definition requires that no non-trivial narrowing steps are taken.

The staged specialization of an operation is built on staged simplification. It is specified by the function \( OP^2_n \), which applies an \( n \)-ary operator to \( n \) arguments and simplifies the result.

\[ OP^2_n \colon \{n : \text{N} \} \Rightarrow \Theta(n) \Rightarrow \text{P} \Rightarrow \text{Def} \times \text{Def} \times P \]
\[ OP^2_n(f)(p_1, \ldots, p_n) = ST^2_P(f(p_1, \ldots, p_n)) \]

### 6. From Binding Times to Cogen

A standard offline specializer cannot easily take advantage of the additional information generated by the analysis proposed in Section 4. Here is why. Suppose the binding time at a program point is \( 1 \cup X \). That means, a values arriving there at specialization time is either a static 1 or a dynamic variable. As both values require a different treatment, the specializer cannot take advantage of the binding-time information but has to inspect the arriving value, anyway.

### 6.1 Attempt: An Offline Specializer

Figure 14 contains a sketch of the definition of such a specializer. Its definition serves to highlight the problems arising with a (generic) specializer. It can be seen as a motivation and a stepping stone towards the construction of a handwritten compiler generator, a cogen, which can use specialized value representations derived from the analysis and thus use a compiled version of the above inspection.

The specializer is written in monadic style in Haskell-like pseudocode. The monad supports all operations that are necessary for specialization: name generation, administration of generated code, and insertion of let expressions to bind generated code to dynamic variables.

The cases for constants, variables, and let expressions are unsurprising. In the case for the function call, the argument patterns have to be coerced to the patterns expected by the function. The notation \( v : p \rightarrow p' \) is used to express the coercion from \( p \) to \( p' \). In the case for the conditional, the function "static" checks if a pattern is fully static. Furthermore, in the static case, the return patterns of the branches, \( p_1 \) and \( p_2 \), have to be coerced to their least upper bound \( p_1 \sqcup p_2 \). If the condition is dynamic, then the results of all branches have to be converted to pure code and the conditional has to be generated.

The most interesting case deals with primitive operations. It first evaluates the arguments and then checks them against all possible combinations of patterns. Then, rewriting is used to obtain an optimal simplification of the application of the operation as explained in Section 2. The rewriting executes the specialization-time operations and the code in \( d \) must be generated. Finally, the return pattern must be instantiated from the computed values and coerced to the expected return pattern. Essentially, the online specializer works in the same way and there is no visible advantage for the offline case because the static computation still proceeds by term rewriting and the knowledge about the specific patterns is not exploited.

However, when generating a cogen from the result of the binding-time analysis, then further stages is possible. First, the patterns showing up in the analysis can be materialized into algebraic datatypes which then form the basis for a transformation of the analyzed program to a generating extension. The algebraic datatypes come with coercion functions which are inserted at join points in the dataflow, i.e., on the branches of conditionals and at function calls.

Second, the staging of the operations can be compiled. The staged rewriting can be executed at cogen time, then the generated specializer executes the static definitions and generates code for the dynamic definitions.

### 6.2 A Compiler Generator (Cogen)

Figure 15 contains an outline of the cogen. That code uses staging annotations for denoting code generation. The annotations are bracket \(< \ldots >\) for code generation and tilde ~ for escaping from code generation, analogously to backquote[17]. The generated code is the customized specializer for a particular source program, a so-called generating extension. The structure of such programs is
well-known, including the fact that environments are not needed, because the binding structure of the underlying source program can be used [3, 9, 13, 18].

The transformation of constants and variables is obvious. The transformation of a let expression does not mention a let expression. If running $C_G[t]$ yields a static value, then the generated specialization binds it to $x$ and continues specialization according to $C_G[t']$. On the other hand, if $C_G[t]$ yields a dynamic value, then it contains only dynamic variables, which are bound in definitions stored in the monad. These definitions are reified into let expressions at the top level and at dynamic conditionals.

The transformation of a function call is a function call in the generated specialization, which has the same structure as the underlying source program. The arguments are specialized, but the patterns have to be adapted to fit the patterns expected by the function (which may be called from several sites). There are well-known techniques for extracting the dynamic parts of a pattern because only these parts needs to be passed as arguments [5, 19].

The transformation of the conditional checks at generation time whether the condition is static. If so, it generates static specialization code, a specialization-time conditional. If the condition is not static, then it generates code for a dynamic conditional: the monadic coercions to dynamic not static, then it generates code for a dynamic conditional: the

---

**Figure 14.** Offline specialization with PSO.

---

**Figure 15.** Offline cogen with PSO.

---

\[
SP[c \mid c \in F] \rho = \\
\text{return } c \\
SP[x] \rho = x \\
SP[let \; x = t \; \text{in} \; t'] \rho = \\
do \; v \leftarrow SP[t] \rho \\
SP[t'] \rho \mid \{x \mapsto v\} \\
SP[F(t_1 : p_1, \ldots, t_n : p_n)] \rho = \\
do \; v_1 \leftarrow SP[t_1] \rho \\
\ldots \\
v_0 \leftarrow SP[t_0] \rho \\
SP[t] \rho \mid \{x \mapsto v, \ldots, v_i \mapsto v_i'\} \\

\text{where } \\
F : (p_1', \ldots, p_n') \to p \\
F(x_1, \ldots, x_n) = t \\
SP[if \; (t_0 : p_0) \; (t_1 : p_1) \; (t_2 : p_2)] \rho = \\
do \; v_0 \leftarrow SP[t_0] \rho \\
if \text{static}(p_0) \text{ then} \\
\text{if } v_0 \text{ then } v \leftarrow SP[t_1] \rho \\
\text{return } (v : p_1 \leadsto p_1 \sqcup p_2) \\
\text{else } v \leftarrow SP[t_2] \rho \\
\text{return } (v : p_2 \leadsto p_1 \sqcup p_2) \\
\text{else do } d_0 \leftarrow v_0 : p_1 \leadsto X \\
d_1 \leftarrow SP[t_1] \rho : p_1 \leadsto X \\
d_2 \leftarrow SP[t_2] \rho : p_2 \leadsto X \\
z \leftarrow \text{gen}(d_0, d_1) \\
\text{return } z \\
SP[f(t_1, \ldots, t_n) : p] \rho = \\
do \; v_1 \leftarrow SP[t_1 : p_1 \sqcup \cdots \sqcup p_{1k_1}] \rho \\
\ldots \\
v_n \leftarrow SP[t_n : p_{1n} \sqcup \cdots \sqcup p_{kn}] \rho \\
case \{(v_1, \ldots, v_n) \text{ of} \\
(p_1'_{i_1}, \ldots, p_{1n}) \to \\
\text{run rewriting to specialize operation} \\
(d, p) = \text{OP}_n(\{ p_1', \ldots, p_{1n} \}) \\
generate \; d \\
\text{return } (p : p \leadsto p_0) \\
\}
\]

\[
CG[c \mid c \in F] = \\
\text{return } "c" \\
CG[x] = \\
\text{return } "x" \\
CG[let \; x = t \; \text{in} \; t'] = \\
\text{do } x \leftarrow "(CG[t])" \\
\text{do } v \leftarrow "(CG[t'])" \\
CG[F(t_1 : p_1, \ldots, t_n : p_n)] = \\
\text{do } v_1 \leftarrow "(CG[t_1])" \\
\ldots \\
v_n \leftarrow "(CG[t_n])" \\
F(v_1 : p_1 \leadsto p_1', \ldots, v_n : p_n \leadsto p_n') > \\
\text{where} \\
F : (p_1', \ldots, p_n') \to p \\
F(x_1, \ldots, x_n) = t \\
CG[if \; (t_0 : p_0) \; (t_1 : p_1) \; (t_2 : p_2)] = \\
\text{do } v_0 \leftarrow "(CG[t_0])" \\
\text{(if static}(p_0) \text{ then} \\
\text{if } v_0 \text{ then } v \leftarrow "(CG[t_1])" \\
\text{return } (v : p_1 \leadsto (p_1 \sqcup p_2)) \\
\text{else } v \leftarrow "(CG[t_2])" \\
\text{return } (v : p_2 \leadsto (p_1 \sqcup p_2)) > \\
\text{else do } d_0 \leftarrow v_0 : p_1 \leadsto X \\
d_1 \leftarrow "(CG[t_1])" : p_1 \leadsto X \\
d_2 \leftarrow "(CG[t_2])" : p_2 \leadsto X \\
z \leftarrow \text{gen}(d_0, d_1, d_2) \\
\text{return } z \\
CG[f(t_1, \ldots, t_n) : p] = \\
\text{do } v_1 \leftarrow "(CG[t_1] : p_1' \sqcup \cdots \sqcup p_{1k_1})" \\
\ldots \\
v_n \leftarrow "(CG[t_n] : p_{1n}' \sqcup \cdots \sqcup p_{kn})" \\
case \{(v_1, \ldots, v_n) \text{ of} \\
(p_1'_{i_1}, \ldots, p_{1n}) \to \\
\text{run staged rewriting to specialize } f \\
"((d_0, d_1, p) = \text{OP}_n(\{ p_1', \ldots, p_{1n} \}) \\
\text{do } uses \; vars \; bound \; in \; p_1' \\
\text{do } binds \; vars \; used \; in \; d_1 \text{ and } p \\
\text{do } binds \; vars \; used \; in \; p \\
\text{< "do } \text{ evaluate } d_0 \text{ at specialization time} \\
\text{gen}(d_1) \text{ generate code that generates code} \\
\text{return } (p : p \leadsto p_0) > \\
\ldots \text{ further cases} \}
\]
list of dynamic definitions $d_1$, and a pattern $p$. This computation is performed at compiler-generation time. The static definitions $d_0$ are then spliced into the generated code, such that they are executed at specialization time. The dynamic definitions $d_1$ are set up for code generation. Finally, the returned pattern $p$ of the specific instance of the operation needs to be coerced to the expected pattern $p_0$. Mostly, this coercion is just an injection.

7. Implementation and Status

We implemented a prototype of the online specializer in Haskell. This prototype implementation can handle all examples in Sections 2 and 3. It further handles the generation of optimized code for the fast Fourier transformation using ideas presented by Kise-lyov and Taha [7, 8].

To test the feasibility of handwriting a compiler generator, we manually generated a generating extension according to the guidelines explained in Section 6. Figure 16 contains the hand-generated code, which is explained in the next few paragraphs. Executing it performs exactly the same specialization as indicated at the end of Section 2. The entry point is $\text{power\_gen}$ and the program operates in the $\mathsf{Gen}$ monad which includes name generation and the context monad $\mathsf{let\_insert}$.

The functions $\text{mlift1}$ and $\text{mlift2}$ lift unary and binary monadic functions to accept monadic parameters. The types $\mathsf{T1}$, $\mathsf{T2}$, and $\mathsf{T3}$ are generated from the patterns arising in the binding-time analysis. The two coercion functions inject $\mathsf{T1}$ and $\mathsf{T2}$ values into $\mathsf{T3}$. The multiplication $\text{mult}$ checks its second argument (of type $\mathsf{T3}$) for representing the constant 1, in which case the first argument is returned. Otherwise, if the second argument is a dynamic value $y$ it generates a multiplication operation. The auxiliary function $\text{let\_insert}$ creates a floating definition for a fresh, dynamic variable and returns this fresh variable. The $\mathsf{Gen}$ monad contains a mechanism for collecting the generated definitions.

The main generator function, $\text{power\_gen}$ is pretty much the transcription of the binding-time annotated program. The condition is fully static, hence its computation and the conditional are directly encoded in the target language. The first branch of the conditional returns the coerced representation of the static 1, whereas the second branch performs the recursive call, uses the specialized multiplication to multiply with $x$, and coerces the resulting code to type $\mathsf{T3}$. This code could be further optimized, for example $\text{coerce\_T1\_T3}$ $\mathsf{T1}$ could be simplified to return $\mathsf{T1}$ and the coercion $\text{coerce\_T2\_T3}$ could be fused into $\text{mult}$, but these optimizations are in scope of the Haskell compiler’s code generator, so that further transformation seems futile.

8. Related Work

Much of the related work in the partial evaluation community has been mentioned and discussed along the way in Section 1. Particularly inspiring was the work by Kise-lyov, Taha, and Swadi [7, 8] on generating optimized code for the fast Fourier transformation. With our online implementation, we have checked that this kind of optimization can be automated, but we have yet to specify and generate the corresponding pattern-based binding-time analysis and the corresponding compiler generator.

Antoy and Hanus [2] propose an extension of functional-programming with function patterns. In this extension, the left-hand sides of a function definition are not restricted to constructor patterns, but they may contain previously defined functions as well. At first sight, our implementation of operations appears related. A closer look shows that, in fact, our patterns are reified arithmetic terms that could be viewed as partially static datastructures and handled by a sufficiently powerful specializer.

Püschel and coworkers [12] consider the automatic generation of highly optimized software libraries for DSP transformations, like the fast Fourier transformation. Their approach is fully transformational. They make the entire definition available to the rewriting and optimization engine to derive highly optimized code. Our approach limits the rewriting and is therefore able to stage it and to optimize the generation time. Furthermore, our approach is compositional so that global transformation cannot be expressed.

Rompf and Odersky [14] have developed an approach to staging that is supported by a similar code generation technique as our generation monad. However, their intermediate representation is graph-based whereas ours consists of a linear list of definitions. It would be interesting to change to a graph-based representation because it makes sharing more explicit and it can elide useless and one-shot definitions while during linearization. In collaboration with others, they used this technique to enable building a modular, user-configurable optimization pipeline for implementing DSLs [15]. The idea here is to define language layers at different levels of abstraction, perform lightweight equation-based optimization at each level, and then expand to the next lower level to continue. The machinery on each of their levels is similar to our optimized operations. However, their goal is not to create efficient program generators, but rather to build customizable optimizing compilers.

9. Conclusions and Future Work

We defined an extension of partial evaluation with partially static operations. This extension enables the specializer to rearrange arithmetic expressions at specialization time according to arithmetic laws. This extension requires a generalization of binding-time analysis to a pattern analysis, where patterns comprise arithmetic operations, constants and dynamic data. We demonstrated that this extension automates good specialization results which previously required a lot of manual work [7, 8].

This paper mostly outlines the theoretical foundations. It states the staging property of the specializer, proves the correctness of the binding-time analysis, and proves the correctness and completeness of a version of the analysis with respect to the online specializer. It further contains the outline of an offline specializer and the specification of a compiler generator all based on the pattern-driven binding-time analysis.

A preliminary account of the ideas in this paper was presented as an invited presentation at PEPM 2013 accompanied by a two-page abstract, which contains no technical material [21].

We plan to implement the binding-time analysis and the compiler generator to perform further experiments and to study the relative performance of the different specialization approaches and apply the system to further examples. We further plan to investigate means to integrate non-trivial narrowing into the compiler generator. Clearly, the substitutions can be integrated into the compiled patterns, but the details of this integration are yet to be determined.

Acknowledgments

Many thanks to Elvira Albert, Olivier Danvy, Martin Elsman, Fritz Henglein, John Hughes, Oleg Kise-lyov, Tia-Rompf, and Martin Sulzmann for discussions, comments, and suggestions.

References

-- monadic lift operation
mlift1 :: Monad m => (a -> m c) -> (m a -> m c)
mlift2 :: Monad m => (a -> b -> m c) -> (m a -> m b -> m c)

-- datatypes from patterns
data T1 = T1 -- number one
type T2 = Ident -- dynamic variable
data T3 = T3_1 | T3_2 Ident -- one or variable

-- coercions
coerce_T1_T3 :: T1 -> Gen T3
coerce_T1_T3 T1 = return T3_1
coerce_T2_T3 :: T2 -> Gen T3
coerce_T2_T3 x = return (T3_2 x)

-- specialized operations
mult :: T2 -> T3 -> Gen T2
mult x T3_1 = return x
mult x (T3_2 y) = let_insert (Op2 MUL (Var x) (Var y))

-- generating extension
power_gen x n =
  if n == 0 then
    coerce_T1_T3 T1
  else
    mlift1 coerce_T2_T3
        (mlift2 mult (return x) (power_gen x (n - 1)))

-- Figure 16. Example program generator for power.


