Program Adaptation via Output-Constraint Specialization *

Siau-Cheng Khoo and Kun Shi
School of Computing
National University of Singapore
{khoosc,shik}@comp.nus.edu.sg

Abstract. In component-based software development, gluing of two software components is usually achieved by defining an interface specification, and creating wrappers on components to support the interface. We believe that interface specification provides useful information for specializing components. An interface may define constraints on a component's inputs, as well as on its outputs. In this paper, we propose a new approach to program specialization with respect to output constraints. We provide the form in which an efficient specialized program should be after such specialization, and consider a variant of partial evaluation to achieve it. In the process, we translate an output constraint into a characterization function for a component's input, and define a specializer that uses this characterization to guide the specialization process. We believe this work will broaden the scope of program specialization, and provide a framework for building more generic and versatile program adaptation techniques.
Categories and subject descriptors: D.3.4 [Programming Languages]: Processors – optimization; I.2.2 [Artificial Intelligence]: Automatic Programming – Program transformation; D.2.13 [Software Engineering]: Reusable Software.
General Terms: Algorithms, Performance.
Keywords: Partial Evaluation, specialization, weakest pre-condition.

1. Introduction

In the last three decades, partial evaluation (PE) has widely been used in many applications, showing remarkable results in improving program efficiency. We have also witnessed various techniques related to this transformation, each attempting to either improve the efficiency of the partial evaluation process or to broaden its scope of application.

Regardless of the improvements and variations, partial evaluation inevitably aims at specializing a program with respect to some aspects of the program's input, and it is with this objective in mind that partial evaluation is also termed *projection* [14].

* This is an expanded version, with major revisions, of the paper entitled "Output-Constraint Specialization", which was presented in the 2002 ACM SIGPLAN ASIA Symposium on Partial Evaluation and Semantics-Based Program Manipulation.

There are times when we wish to specialize a program, the context of which does not conveniently reveal to us its input constraint. Consider a component-based program development, in which software components are reused to construct programs. Gluing of software components requires clear definitions of interfaces acting as contracts between them. It is not uncommon to find that the existing components involved are too general for a particular interface at hand.

A common technique used in component composition is by creating wrappers (of the components) with the aim of adapting the output of one component to the input of the other. We call this the *wrapper approach.* While such wrappers can be automatically generated, their use increases the run-time overhead of the combined system.

A more desirable solution would be to specialize the components according to the contract specified by the interface. Consider the case when data produced by a component A is passed to another component B, adhering to the interface's specification. For component B, the interface contract provides the necessary input constraint (called a pre-condition), with respect to which component B can be partially evaluated. Here, a constraint-based partial evaluator such as that developed by Lafave *et al.* [17] suits the task well. However, for component A, the contract specifies its necessary *output* condition (called a post-condition). The specialization of component A cannot be realized by traditional partial evaluation techniques.

It is also desirable to adapt a stand-alone program to some output constraint. Consider a program that simulates the behaviour of a vending machine as follows: It can take in $10c$, $20c$, $50c$, and \$1 coins, and it sells coffee($80c$), green tea($60c$), black tea($60c$), and coke(\$1). Suppose we wish to modify the program to simulate the behaviour of another vending machine which only dispenses tea, a plausible solution is to specialize the original program with respect to the output constraint.

Declaratively, an output-constraint specialization (**OCS**) can be defined as follows:

DEFINITION 1 (Output-Constraint Specialization). *Given a program with a main function $f(x_1, x_2, \ldots, x_n) = e$, and a constraint Φ about the function's output, an* output-constraint specialization *will produce a new program with main function f' such that, for a given input tuple $(c_1, c_2, \ldots, c_n)$, if y is a non-bottom result of the application $f(c_1, c_2, \ldots, c_n)$, then*

- *if y satisfies Φ, then $f'(c_1, c_2, \ldots, c_n) = y$.*
- *if y does not satisfy Φ, then $f'(c_1, c_2, \ldots, c_n) = Error$, where $Error$ can be represented by some error messages.*

A naive way to achieve the effect of output-constraint specialization is to add a check to f's output, as follows:

$$f'(x_1, x_2, \ldots, x_n) = \textbf{if } (y \text{ satisfies } \Phi) \textbf{ then } y \textbf{ else } \mathit{Error} \qquad \text{where } y = f(x_1, x_2, \ldots, x_n). \tag{1}$$

This approach is no different from the wrapper approach described earlier. The new program will run slightly slower than the original one.

A more desirable specialized program should possess the following characteristics:

- For input leading to the desired output (*ie.*, satisfying the output constraint), it can produce the same output at possibly reduced computation cost.
- Otherwise, it aborts the computation as early as possible.

It is not always possible to maximize both goals at the same time in practice. Some trade-off must be made.

Sometimes, reduced computation cost in specialized program cannot be ensured if conventional partial evaluation techniques are used for output-constraint specialization. For instance, given the following function definition:

$$f(m, n) \;=\; \textbf{if } (m > 0) \textbf{ then } m + n \textbf{ else } n$$

Suppose that we wish to specialize f such that the output is $3 + n$, a simple calculation will reveal that this can happen only when the input m takes the value 3. Thus, a reasonably efficient specialized program will be as follows:

$$f(m, n) \;=\; \textbf{if } (m = 3) \textbf{ then } 3 + n \textbf{ else } \mathit{Error} \tag{2}$$

Here, the efficiency arises from the introduction of a new test $(m = 3)$ in the specialized program. Such introductions are never done in conventional partial evaluation.

In this paper, we propose a systematic approach for performing output-constraint specialization (abbreviated as OCS). Specifically, given a program with an output constraint Φ, we identify a class of program inputs which will lead to (if the computation ever terminates) an output satisfying Φ, and another class of inputs which will *not* lead to an output satisfying Φ. We then specialize the program with respect to these two classes of inputs. Our main technical contribution is to derive a means to *characterize program inputs by program outputs, and a sufficient condition to detect the existence of a best characterization of program inputs.*

As we have mentioned earlier, a conventional partial evaluator, even a constraint-based partial evaluator, may not be adequate for attaining the desired specialization. We thus define a variant of partial evaluation to accomplish the task.

While our solution leverages on existing techniques in program analysis and semantics-based transformation, it enables us to broaden the scope of program specialization.

The outline of this paper is as follows: In Section 2, we review the background of this work, including the language syntax and the constraint system used. In Section 3, the general specialization approach is explained, together with a close examination of some of the issues involved. This is followed by Section 4, which examines in detail the use of weakest pre-condition in practice to relate input constraints to output constraints. Section 5 describes the analysis phase in detail, and Section 6 illustrates an off-line output-constraint specialization process. We discuss related work in Section 7, before concluding in Section 8.

2. Language and Constraints

We apply our technique to a simply typed first-order functional language with strict semantics. The language is defined in Figure 1. For ease of presentation, we restrict function definitions to top-level ones, and only allow variables to be passed as arguments to functions. These restrictions do not reduce the generality of our method, as transformations exist (such as the let-abstraction) to convert an ordinary program to one conforming to the restriction.

A program consists of a list of function definitions and possibly some data-type declarations. One of the definitions is the main function. Program execution begins by passing some arguments to the main function. These arguments are also called *program inputs*. For simplicity, the only data-type declaration we introduce is the enumerated data type.

Constants in the language include integers, boolean, and enumerated data. The following is a valid function in our language:

$$
\begin{array}{l}
\mathit{data\ Beverage} \;=\; \mathit{Coke} \;\mid\; \mathit{GreenTea} \;\mid\; \mathit{BlackTea} \;\mid\; \mathit{Coffee} \\
\mathit{choose} \;::\; \mathit{Int} \;\rightarrow\; \mathit{Beverage} \\
\mathit{choose}\ n \;=\; \begin{array}[t]{l}
\mathbf{if}\ (n == 1)\ \mathbf{then}\ \mathit{Coke} \\
\mathbf{else\ if}\ (n == 2)\ \mathbf{then}\ \mathit{GreenTea} \\
\mathbf{else\ if}\ (n == 3)\ \mathbf{then}\ \mathit{BlackTea} \\
\mathbf{else}\ \mathit{Coffee}
\end{array}
\end{array}
$$

Functions (and expressions) can be annotated with *constraints* describing the relationship between their input (or free variables) and output.

$$
\begin{array}{lllr}
x & \in & \mathbf{Var} & \langle\text{Variables}\rangle \\
f & \in & \mathbf{Prim} + \mathbf{FName} & \langle\text{Function Names}\rangle \\
c & \in & \mathbf{Const} & \langle\text{Constants}\rangle \\
e & \in & \mathbf{Exp} & \langle\text{Expressions}\rangle \\
 & & e ::= x \mid c \mid \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 \mid & \\
 & & \quad\quad \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid f\ (x_1, \ldots, x_n) & \\
d & \in & \mathbf{Decl} & \langle\text{Definitions}\rangle \\
 & & d ::= f\ (x_1,\ \ldots,\ x_n) = e &
\end{array}
$$

Figure 1. The Language Syntax

A constraint can be viewed as an assertion about the behaviour of a function, or the property of a piece of data. In this paper, we call it a *size constraint*. It can aid in the derivation of constraints needed for specialization.

Constraints are formulated in the framework of the sized-type system as proposed by Chin and Khoo [4, 5]. However, we wish to point out that our proposed technique works equally well with other constraint systems, such as dependent-type systems [29], and verification conditions proposed by Flanagan [13] – which solve constraints via a theorem prover.

A sized-type system aims at capturing and propagating size information of a program. Size information is relational in nature, where an interdependency the relationship amongst inputs, as well as relationship between input and output may be captured by linear constraints. The size information for an expression describes the relationship between the free variables and the expected result of the expression. The sized type of each expression e can be expressed by $e :: (\tau,\ \phi)$, where τ is an annotated sized type containing size variables, and ϕ is a constraint expressed in terms of the size variables in τ. While what constitutes size information is subject to interpretation, we take as default the following:

- The size of an integer is the integer value itself. *Eg.* $3 :: (\mathbf{Int}^v,\ v = 3)$.

- The size of a boolean value is either 0 (for *False*) or 1 (for *True*). Boolean is an instance of enumerated types. In general we encode each data constructor in an enumerated type by an integer. For instance, given the declaration of data type *Beverage* given above, we can assign 0 to *Coke*, 1 to *GreenTea*, and so on.

– The size of a variable is a relation specifying the sizes of the possible values associated with the variable.

– The size of a function is a relation between the sizes of its input and output.

We can now provide the sized type of the above *choose* function definition as follows:

$$\begin{array}{l} \textit{choose} :: (\mathbf{Int}^n \rightarrow \textit{Beverage}^r, \\ \quad (0 \leq r \leq 3) \wedge \\ \quad ((n=1 \wedge r=0) \vee (n=2 \wedge r=1) \\ \quad \vee (n=3 \wedge r=2) \vee \\ \quad (\neg(n=1 \vee n=2 \vee n=3) \wedge r=3))) \end{array}$$

Throughout this paper, we *reserve variable* r *as the size variable capturing the returned values of a function.*

For the sake of efficient computation, size relations are expressed in terms of presburger formulae [12], and computed with the help of some efficient constraint-solving technology (*e.g.* Pugh's Omega Calculator [21].) Presburger formulae can be built from affine constraints over integer variables (*eg.* $r = 2 * v$ is a valid Presburger formula, but $r = v * v$ is not), the logical connectives $\neg$, $\wedge$, $\vee$, and the quantifiers $\exists$ and $\forall$ ($\forall$ is viewed as an abbreviation of $(\neg \exists \neg)$).

The syntax of sized types is depicted in Figure 2. A formula can be of the form $[v_1, \ldots, v_m] \rightarrow [w_1, \ldots, w_n] : \Phi$. This specifies a relation Φ between the size variables of a function's input ($[v_1, \ldots, v_m]$) and the size variables of the function's output ($[w_1, \ldots, w_n]$). For example, the size of the *choose* function can be re-written as:

$$\begin{array}{l} [n] \rightarrow [r] : (0 \leq r \leq 3) \wedge \\ \quad ((n=1 \wedge r=0) \vee (n=2 \wedge r=1) \\ \quad \vee (n=3 \wedge r=2) \vee \\ \quad (\neg(n=1 \vee n=2 \vee n=3) \wedge r=3)) \end{array}$$

As the formula is relational, we can also express the formula in *set-like format*, as in $[n, r] : \cdots$.

For an annotated type, we retrieve its size variables through the fv function. It is defined as follows:

$$\begin{array}{rclrcl} fv(Int^v) & = & \{v\} & fv(\tau_1 \rightarrow \tau_2) & = & fv(\tau_1) \cup fv(\tau_2) \\ fv(Bool^v) & = & \{v\} & fv(\tau_1, \ldots, \tau_n) & = & \bigcup_{i=1}^{n} \{fv(\tau_i)\} \end{array}$$

Concerning notation, when $X = \{v_1, \ldots, v_n\}$ is a set of size variables, we write $\exists X . \phi$ or $\exists v_1 \ldots v_n . \phi$ as a shorthand for $\exists v_1 . \ldots . \exists v_n . \phi$.

Sized Type = (**AnnType**, **F**)

Annotated Type Expressions:

$v,\ w \in \mathbf{V}$ ⟨Size Variables⟩
$\tau \in \mathbf{AnnType}$ ⟨Annotated Types⟩
$\tau ::= b \mid (\tau_1, \ldots, \tau_n) \mid \tau_1 \to \tau_2$
$b \in \mathbf{Basic}$ ⟨Basic Types⟩
$b ::= \mathbf{Int}^v \mid \mathbf{Bool}^v \mid \cdots$

Formulae:

$\Psi \in \mathbf{F}$ ⟨Formulae⟩
$\Psi ::= \Phi \mid [v_1, \ldots, v_m] : \Phi \mid [v_1, \ldots, v_m] \to [w_1, \ldots, w_n] : \Phi$
$\Phi ::= \phi \mid \neg\, \Phi \mid \exists\, v \,.\, \Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2$

Size Formulae:

$\phi \in \mathbf{Fb}$ ⟨Boolean Expressions⟩
$\phi ::= \mathit{True} \mid \mathit{False} \mid a_1 = a_2 \mid a_1 \neq a_2$
$\mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2$
$a \in \mathbf{AExp}$ ⟨Arithmetic Expressions⟩
$a ::= n \mid v \mid n * a \mid a_1 + a_2 \mid -\, a$
$n \in \mathcal{Z}$ ⟨Integer Constants⟩

Figure 2. Syntax of Sized Types

3. Issues in Output-Constraint Specialization

In a naive version of OCS, such as the function defined in equation (1), the specialized program will subject all its outputs to a test on output constraints. Consequently, the specialized program always runs slower than the original program. As we have mentioned, it is no different than wrapping the program by an interface. Nevertheless, this naive approach sets a lower bound to the efficiency of the specialized programs we aim to attain.

A more promising approach to this problem is to replace as many as possible tests on output constraints by tests on input constraints. Specifically, this approach identifies an input constraint such that satisfiability of this input constraint implies satisfiability of the output constraint. The idea here is to raise the tests to the beginning of the program, so that decisions (especially the decision to abort) can be made as early as possible. Equation (2) is the result of such an approach. The benefit of including input tests in the specialized program is that: we can aggressively specialize the portion of the program which is guaranteed to lead to an output satisfying the output constraint.

In the perfect case when the input constraint thus identified encompasses all possible inputs leading to the desired output, we can also immediately abort the computation for those inputs that *do not* satisfy the input constraint, thus avoiding useless computation. Consequently, the resulting specialized program takes the following form (which we call the *perfect* form):

$$f\ n \ = \ \begin{array}{l} \textbf{if}\ \text{(input satisfying the input constraint)}\ \textbf{then} \\ \quad < \text{specialized code without output constraint test} > \\ \textbf{else}\ \mathit{Error} \end{array} \qquad (3)$$

There are potential limitations to this approach:

1. It can be problematic to include an input constraint as a piece of code in the specialized program, since such constraint, in order to be precise enough, may not be easily coded. Moreover, once coded, its computational complexity may be too expensive. For example, the input constraint may be a primality test, *etc.* Such a check may incur more run-time cost than a simple specialized program obtained by the wrapper approach. Thus, in practice, the run-time cost of executing an input constraint needs to be weighed against the saving gained from both aggressive specialization and omission of the output-constraint test.

2. To reap the full benefit of this approach – and thus obtain specialized programs akin to the perfect form (3), we should partition inputs cleanly into two categories: one category contains inputs that must lead to the desired output (and the corresponding code can thus be aggressively specialized), and the other contains those that *must not* lead to the desired output (and the corresponding code can thus be replaced by an *Error* message, signifying abort operation.) However, it is not obvious whether such a clean partition of input can be attained.

We believe that the first limitation above is, in most situations, inevitable in our pursuit to replace output-constraint tests by tests over other program variables, including program inputs. Thus, the decision to introduce any input-constraint test is mainly an engineering issue. While this issue is worth investigating, it is beyond the scope of this paper.

On the other hand, the second limitation can be partially overcome through careful and innovative derivation of input constraints. It is therefore the main focus of this paper. Specifically, we shall describe in detail the inference of input constraints from output constraints, and provide a sufficient condition for clean partitioning of program inputs.

Our approach to output-constraint specialization comprises two phases: an analysis and a specialization phase. The analysis phase takes in a program and an output constraint. It derives two input constraints for the main function (and the other functions too, for the reason to be given in Section 6). These two input constraints attempt to divide the input domain into two sets, as mentioned in the discussion for limitation 2 above. When such division results in a clean partitioning of program inputs, we can obtain a specialized program resembling the perfect form. Otherwise, we have at least two options for generating the specialized program.

In the first option, we may generate the specialized program in the form (4) below:

$$\begin{array}{lll} f\ n & = & \textbf{if}\ \text{(input leading to desired output)}\ \textbf{then} \\ & & \quad < \text{specialized code without output constraint test} > \\ & & \textbf{else if}\ \text{(input leading to undesired output)}\ \textbf{then}\ \textit{Error} \\ & & \textbf{else}\ \ < \text{wrap code in output constraint test} > \end{array} \tag{4}$$

We name this form the *double-test*, as two input constraints are included in the specialized programs. At the last branch, we are not able to determine if an input does lead to the desired output, neither are we able to determine that it does *not* lead to the desired output. Hence, we simply replace the branch by the original program wrapped with the output-constraint test. Though simple, this form may not be appealing because of the inclusion of two input-constraint tests.

In the second option, we generate a specialized program that may contain output-constraint tests at some branches of the original program. However, it *does not contain any input-constraint test*. We name this form *no-test*, for the obvious reason that no input-constraint test is involved.

This no-test form of specialized programs has the advantage that it avoids any use of an input-constraint test, while still being able to produce efficient code (without wrapping) for some branches which can only produce a desired output.

In this paper, we adopt the decision to use the perfect form when a clean division of program inputs can be obtained. Otherwise, we choose to specialize the programs into no-test form, as we can then address some of the more interesting specialization issues pertaining to this option. [1]

[1] In our previous paper [16], we described solely the generation of specialized programs in no-test format.

4. Weakest Pre-Condition

Input constraints aim at characterizing program inputs by their ability to generate outputs that satisfy the output constraint. In the theory of program semantics, this problem of deriving the best input constraint leading to the desired output is tackled via the technique of *weakest pre-condition* (**WPC**) derivation.

If, for a given program and an output constraint, we were able to derive its weakest pre-condition, then our OCS would have been much simpler. Specifically, the specialized program would be in perfect form, in which the conditional test would be a check on whether a program input satisfies the weakest pre-condition.

In practice, it is often the case that the derived pre-condition is stronger than WPC. That is, there may exist some program inputs which do not satisfy the derived pre-condition but nevertheless satisfy WPC. Thus, the derived pre-condition cannot be used as a conditional test in the specialized program written in perfect form.

This observation is crucial to the understanding of OCS. *It is not adequate to specialize a program with respect to a subset of input that guarantees to yield the desired output.* Rather, we need to at least ensure that *all* possible inputs leading to the desired output are supported by the specialized program.

Derivation of a pre-condition stronger than WPC can be viewed as a *must*-analysis. Such analysis determines a (possibly proper) subset of program inputs that satisfies WPC. One may wonder if it is desirable to perform a derivation using some *may*-analysis, thus yielding a pre-condition that is *weaker* than WPC. In this case, all valid inputs (*ie.* inputs satisfying the WPC) are included in the resulting pre-condition. However, this attempt is futile, as the derived pre-condition may also include inputs not satisfying the WPC. Therefore, the specializer does not know, among all the inputs satisfying the resulting pre-condition, which inputs do/do not lead to the desired output. Consequently, the specialized program will have to include many output-constraint checks, since it must also be the case that *no output that does not satisfy the output constraint should be produced*.

In summary, the reason that derived pre-conditions may not be used in attaining specialized programs of perfect form is because it is difficult to infer that such a derived pre-condition is indeed WPC.

Note that derivation of WPC assumes termination of program. As such, WPC does *not* include input values which cause a program not to terminate (these inputs are called *non-terminating inputs*). In the presence of programs which may not terminate on some inputs, it becomes difficult to obtain the theoretical WPC automatically, as the problem of termination is in general undecidable. Therefore, we may have to include non-terminating inputs in the result of *WPC* derivation. We call this result a *variant* of *WPC*.

DEFINITION 2 (*WPC*-Variant). *Let w be the WPC of a program with respect to an output constraint. A* variant*, v, of w is defined as follows:*

> *For any input i, if executing the program with input i terminates, then i satisfies w if and only if i satisfies v.*

The above definition states that a *WPC*-variant covers the same set of program inputs as *WPC*, *provided executing the program with those inputs always terminate.* Thus, these two constraints differ only in their inclusion of inputs which do not cause program to terminate during their executions.

How do we determine if a derived pre-condition is a *WPC*-variant? Our idea is to look at "the other side of the problem". Parallel to the derivation of pre-condition from an output constraint, we also employ the same technique to derive a pre-condition from the *negated output constraint*. We call this derived pre-condition a *negative* pre-condition. Correspondingly, the derived pre-condition from the original output constraint is called a *positive* pre-condition. Program inputs that satisfy negative pre-condition are guaranteed to yield undesired output, if their computation terminates. Consequently, code associated with producing undesired outputs should not be generated by OCS.

Briefly, when both positive and negative pre-conditions cover the entire range of program inputs, we will show that the positive pre-condition is a *WPC*-variant, and the specialized program can be written in perfect form. Otherwise, we will express the specialized program in no-test form.

In the following section, we shall describe this derivation process in detail, through an illustration using a sized-type system.

5. The Analysis Phase

We introduce in this section an analysis that infers an input condition of a program associated with an output constraint.

5.1. Forward Contextual Analysis

We perform a forward contextual analysis starting from the main function of a program. It aims to compute the context of the program's output. In general, a context is a formula relating the output of an expression to its input. Such a context is described by a Presburger formula, and called a *contextual constraint*.

Algorithm $\mathcal{C}$ depicted in Figure 3 traverses the syntax tree representing the right-hand side of the main function of a program. It relies on the size information available in the program. Constraints gathered during traversal include constraints for selecting a branch (of **if**-expression), assertion about the sizes of local variables, and post-conditions of function calls, as obtained from the input-output relation of that function and the constraints of its arguments. Prior to this analysis, we assume that size information about individual function definitions is available, either through the execution of a size-type system (as described in [4, 5]) or through user annotation. This includes size information for recursive function definitions. Therefore, it is not necessary for our analysis to re-compute a size information for recursive calls.

$$
\begin{array}{l}
\mathcal{C} \;::\; \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{F} \rightarrow (\mathbf{AnnType} \times \mathbf{F}) \\
\qquad\qquad \textit{where} \;\; \mathbf{Env} = \mathbf{Var} \rightarrow \mathbf{AnnType} \times \mathbf{F} \\
\mathcal{C}[\![x]\!] \; \Gamma\, \psi \;=\; \textit{let} \; (\tau^{v_1}, \phi) = \Gamma [\![x]\!] \\
\qquad\qquad\qquad\quad v = \textit{newVar} \\
\qquad\qquad\quad \textit{in} \; (\tau^{v}, (v = v_1)) \\
\mathcal{C}[\![n]\!] \; \Gamma\, \psi \;=\; \textit{let} \; v = \textit{newVar} \;\; \textit{in} \; (\mathbf{Int}^{v}, (v = n)) \\
\mathcal{C}[\![f \; (x_1, \ldots, x_n)]\!] \; \Gamma\, \psi \;= \\
\quad \textit{let} \; ((\tau_1^{v_1}, \ldots, \tau_n^{v_n}) \rightarrow \tau, \phi_f) = \textit{renameVar} \; (\Gamma [\![f]\!]) \\
\qquad\quad Y = \{v_1, \ldots, v_n\} \\
\qquad\quad (\tau_i'^{v_i'}, \phi_i) = \Gamma [\![x_i]\!] \, \forall \, i \in \{1, \ldots, n\} \\
\qquad\quad \phi = \exists \, Y \, . \; \phi_f \wedge (\wedge_{i=1}^{n} \, (v_i' = v_i)) \\
\quad \textit{in} \; (\tau, \phi) \\
\mathcal{C}[\![\mathbf{if} \; e_0 \; \mathbf{then} \; e_1 \; \mathbf{else} \; e_2]\!] \; \Gamma\, \psi \;= \\
\quad \textit{let} \; (\mathbf{Bool}^{v}, \phi) = \mathcal{C}[\![e_0]\!] \; \Gamma\, \psi \\
\qquad\quad (\tau_1^{v_1}, \phi_1) = \mathcal{C}[\![e_1]\!] \; \Gamma \;\; (\phi \wedge (v = 1) \wedge \psi) \\
\qquad\quad (\tau_2^{v_2}, \phi_2) = \mathcal{C}[\![e_2]\!] \; \Gamma \;\; (\phi \wedge (v = 0) \wedge \psi) \\
\qquad\quad \tau_3^{v_3} = \textit{renameVar} \; (\tau_1^{v_1}) \\
\qquad\quad Y = \{ v, v_1, v_2 \} \\
\qquad\quad \phi_3 = \exists \, Y \, . \, \phi \wedge (((v_1 = v_3) \wedge (v = 1) \wedge \phi_1) \\
\qquad\qquad\qquad\qquad \vee ((v_2 = v_3) \wedge (v = 0) \wedge \phi_2)) \\
\quad \textit{in} \; (\tau_3, \phi_3) \\
\mathcal{C}[\![\mathbf{let} \; x = e_1 \; \mathbf{in} \; e_2]\!] \; \Gamma\, \psi \;= \\
\quad \textit{let} \; (\tau_1, \phi_1) = \mathcal{C}[\![e_1]\!] \; \Gamma\, \psi \\
\qquad\quad (\tau, \phi_2) = \mathcal{C}[\![e_2]\!] \; \Gamma[(\tau_1, \phi_1)/x] \;\; \psi \\
\qquad\quad Y = \textit{fv}(\tau_1) \\
\qquad\quad \phi = \exists \, Y \, . \; \phi_1 \wedge \phi_2 \\
\quad \textit{in} \; (\tau, \phi)
\end{array}
$$

Figure 3. Definition of the Context-Derivation Function $\mathcal{C}$

$\mathcal{C}$ operates on expressions. It takes in a sized-type environment Γ which binds program variables to sized types. It also binds primitives and user-defined functions to their respective input-output relations. It produces a tuple consisting of: the annotated type of the subject expression, and its contextual constraint.

$\mathcal{C}$ also maintains a formula ψ (of type $\mathbf{F}$, as defined in Fig. 2), representing the contextual constraint before analyzing an expression. Initially, ψ is set to some facts (*ie.*, constraints) about the input and output. For example, there are only four values for *Beverage* and if r is the size variable of a parameter (or an expression result) of type *Beverage*, then we must have $0 \leq r \leq 3$. Also, if n is a natural number, then $n \geq 0$. We call these constraints the *inherent constraints* of size variables. In the algorithm $\mathcal{C}$, its initial input ψ for analyzing program *choose* will be $0 \leq r \leq 3$ (there is no constraint on the input.)

Concerning notation, in Figure 3, function *newVar* returns a new (size) variable, *renameVar* performs renaming of size variables (*ie.*, α-conversion). *renameVar* is overloaded so that it can take in either an annotated type or a sized type (which is a pair). It consistently renames all size variables occurring in its argument, by giving them new and unique names. Furthermore, $\Gamma[(\tau, \phi)/x]$ denotes the update of environment Γ by a new binding of sized type (τ, ϕ) to x.

During computation, when a branch of an **if**-expression is chosen, the context of this branch is included in ψ. When a function call is encountered, its contextual constraint is derived from the sized type of the function – with appropriate size-variable renaming for the arguments.

As a simple example, the sized information of the equality function $(x == y)$ can be defined as:

$$((v_1 = v_2) \wedge (v = 1)) \vee (\neg(v_1 = v_2) \wedge v = 0)$$

where v_1, v_2 and v are size variables of x, y and the result of equality, respectively. Given an expression $n = m$, the resulting size information gathered will be:

$$\begin{aligned}((w_1 = w_2) \wedge (w = 1)) \vee (\neg(w_1 = w_2) \wedge w = 0)& \\ \wedge\ w_1 = v_n \wedge w_2 = v_m&\end{aligned}$$

where v_n and v_m are the size variables of n and m respectively, and w_1, w_2, and w are renamed size variables for v_1, v_2, and v respectively.

To complete the process of context computation, we need to existentially quantify the free size variables appearing in the returned formula, except the size variables for input parameters and output. Thus, for the case of the *choose* function, after simplifying the formula, we can express the context as follows:

$$
\begin{aligned}
[n] \;\rightarrow\; [r] \;:\; & (0 \;\leq\; r \;\leq\; 3) \wedge \\
& ((n = 1 \;\wedge\; r = 0) \;\vee\; (n = 2 \;\wedge\; r = 1) \\
& \;\vee\; (n = 3 \;\wedge\; r = 2) \vee \\
& (\neg(n = 1 \;\vee\; n = 2 \;\vee\; n = 3) \;\wedge\; r = 3)))
\end{aligned}
$$

Notice that no special treatment is required for recursive calls. This is because the size information about a function, including a recursive function, has been captured by the input-output relation. The goal of C is simply to utilize this information for collecting contextual information and formulating contextual constraints. We refer the readers to [5] for a detailed description of input-output relationship computation.

5.2. Characterizing Inputs

Given a program P and an output constraint Φ, we would like to characterize as many of P's inputs as possible by their ability to produce output satisfying (or not satisfying) Φ. As mentioned in Section 4, the best way to achieve such characterization is the weakest pre-condition (*WPC*) derivation. Here, the weakest pre-condition is expressed by the term $wpc.P.\Phi$. Function-wise, we can consider *wpc* as a function taking a program and an output constraint, and returning its weakest pre-condition. *WPC* for functional programs can be defined formally by first translating the program P into a program in *Passified Guarded Command Language* (PGCL) form [13]. As PGCL is of imperative nature, we can define weakest pre-condition using the familiar method.

Given a program and its output constraint Φ, the algorithm $\mathcal{C}$, defined in Section 5.1, can be used to compute the weakest pre-condition of the program. This is possible if we treat Φ as an assertion about the program's output. This fact is expressed in the following theorem:

THEOREM 1. **(*WPC* of Functional Programs)** *Given a program P and an assertion Φ about its output. Denote the result of performing $\mathcal{C}$ over P by $Ctx(P)$. Let P' be the corresponding PGCL program translated from P. Then,*

$$wpc.P'.\Phi = \forall X.(Ctx(P) \Rightarrow \Phi).$$

where X contains all free variables in the formula, except the input size variables.

In the theorem, the phrase *performing $\mathcal{C}$ over P* is defined as the following application:

$$
\begin{array}{l}
\mathcal{C}\,[\![\, e_{main} \,]\!]\, \Gamma_{init}\, \phi_{init} \\
\text{where } \; e_{main} \; = \; \text{body of the main function } f_{main} \text{ of } P \\
\qquad\quad \Gamma_{init} \; = \; \Gamma_x \cup \Gamma_f \\
\qquad\quad \Gamma_x \; = \; \{\, (\tau_x, True) \mid x \text{ is parameter of } f_{main} \,\} \\
\qquad\quad \Gamma_f \; = \; \{\, (\tau_f, \phi_f) \mid f \text{ is a function occurring in } P \,\} \\
\qquad\quad \phi_{init} \; = \; \text{initial constraint on parameters of } f_{main}
\end{array}
$$

The proof can be found in Appendix A.

As the PGCL program P' is equivalent to the functional program P, we will simply write *wpc*.$P.\Phi$ instead of *wpc*.$P'.\Phi$ for ease of presentation.

Theoretically, the importance of *WPC* computation is that it includes *all* input values to a program which are guaranteed to produce the desired output. For the case of the *choose* function, let's assume that we wish to restrict the function to only produce *Coffee*. The output constraint is thus $r = 3$. To make the example more interesting, let us further *assume that its input to function choose must be non-negative; ie.* $n \geq 0$. Thus, we obtain the following *wpc* :

$$
\begin{array}{rl}
wpc.choose.(r=3) & = [n] \,:\, \forall\, r \,.\, Ctx(choose) \Rightarrow (r = 3) \\
 & = [n] \,:\, n \leq 0 \vee n \geq 4 \\
\text{where } Ctx(choose) & = (n \geq 4 \wedge r = 3) \vee (n = 1 \wedge r = 0) \vee \\
 & \quad\; (n = 2 \wedge r = 1) \vee (n = 3 \wedge r = 2).
\end{array}
$$

The accuracy of *WPC* computation depends on the accuracy of context information gathered, which is dependent on the constraints associated with each expression in the program. In practice, these constraints may be approximated, yielding a contextual constraint that is weaker than expected. Denoting an approximated result as $Ctx^a(P)$ and the theoretical one as $Ctx^t(P)$, we must have $Ctx^t(P) \Rightarrow Ctx^a(P)$. For instance, consider the following function $g1$:

$$
\begin{array}{rl}
g1\ n \; = & \textbf{if } n \leq 2 \textbf{ then } 1 \\
 & \textbf{else if } n * n \leq 25 \textbf{ then } 2 \textbf{ else } 3
\end{array}
$$

It has a "theoretical" context as follows (assuming that the program input n must be natural number):

$$
\begin{array}{rl}
Ctx^t(g1) = & (0 \leq n \leq 2 \wedge r = 1) \vee (2 < n \leq 5 \wedge r = 2) \\
 & \vee\ (n > 5 \wedge r = 3)
\end{array}
$$

However, the size of the expression $n * n \leq 25$ cannot be expressed in Presburger arithmetic. Consequently, during analysis, the context will be approximated as follows:

$$Ctx^a(g1) = (0 \leq n \leq 2 \wedge r = 1) \vee (n > 2 \wedge (r = 2 \vee r = 3))$$

The computed context shows that the output of $g1$ can be either 2 or 3 when the input n is greater than 2.

This weakening of computed contextual constraint implies that the computed (derived) pre-condition will be stronger than the theoretical *WPC*:

$$pc^a.P.\Phi \stackrel{def}{\equiv} \forall X\,.\,(Ctx^a(P) \Rightarrow \Phi) \Rightarrow \forall X\,.\,(Ctx^t(P) \Rightarrow \Phi)$$

For the case of function $g1$, let Φ be $r = 3$. We have:[2]

$$\begin{array}{lcl} wpc.g1.(r=3) & = & [n] : n \leq -1 \vee n > 5 \\ pc^a.g1.(r=3) & = & [n] : n \leq -1 \end{array}$$

The result of pc^a indicates that no program input of natural numbers can produce the desired output — certainly a very strong pre-condition.

As pc^a is stronger than the *wpc*, we cannot be certain if an input that fails to satisfy $pc^a.P.\Phi$ will cause the program P not to produce the desired output.

We next turn to the identification of program inputs which do *not* lead to desired outputs. As mentioned in Section 4, this set of inputs can be captured by computing the weakest pre-condition of a program P with respect to a *negated* output constraint, $\neg\Phi$. For function $g1$, we have

$$\begin{array}{lcl} wpc.g1.(r \neq 3) & = & \forall X\,.\,Ctx^t(g1) \Rightarrow (r \neq 3) = [n] : n \leq 5 \\ pc^a.g1.(r \neq 3) & = & \forall X\,.\,Ctx^a(g1) \Rightarrow (r \neq 3) = [n] : n \leq 2 \end{array}$$

Although all program inputs that are less than or equal to 5 do not lead to the desired output $(r = 3)$, the actual derived pre-condition only detects those values which are less than or equal to 2.

Lastly, we differentiate the two weakest pre-conditions, $wpc.P.\Phi$ and $wpc.P.(\neg\Phi)$ by calling them *positive* weakest pre-condition, and *negative* weakest pre-condition, respectively. Likewise, we have positive and negative derived pre-conditions.

---

[2] Given the initial assumption that program inputs are naturals $(n \geq 0)$, the derived pre-condition implies that no program input can produce the desired output. Informally, this is the same as saying that the pre-condition is *False*. The reason that the derived constraint is not just *False* is because the Presburger simplification takes as its working domain the entire integer set. We will remedy this in the later section.

5.3. Full Input Characterization

When all the program's inputs can be precisely characterized by its ability to produce (or not to produce) the desired output, modulo termination, we say that the input domain has been *fully characterized*. More formally,

DEFINITION 3 (Full Characterization). *Given a program P and an output constraint Φ, let Ψ_+ and Ψ_- be the positive and negative pre-conditions of P with respect to Φ. We say that* the pair $(\Psi_+, \ \Psi_-)$ fully characterizes the inputs of P with respect to Φ *if for any program input i, if execution of P with input i terminates and returns a result r, then the following holds:*

1. *i satisfies either Ψ_+ or Ψ_-;*
2. *if i satisfies Ψ_+, then r satisfies Φ;*
3. *if i satisfies Ψ_-, then r satisfies $\neg \Phi$;*

Furthermore, we call the set of program inputs which ensures termination of the execution of P the terminating inputs.

A consequence of full characterization is that the set of terminating inputs can be partitioned into two disjoint sets, one satisfying Ψ_+, and the other satisfying Ψ_-. Consequently, the specified function can be specialized into the perfect form.

Because of their roots in program semantics, the positive and negative *WPC*'s of a program (with respect to an output constraint) is an obvious pair of input constraints fully characterizing the program inputs. We state this formally as a property about *WPC*:

PROPERTY 2. *Given a program P and an output constraint Φ, the following pair of weakest pre-conditions*

$$wpc.P.\Phi \quad \text{and} \quad wpc.P.\neg\Phi$$

fully characterizes the input of P with respect to Φ.

For the function $g1$ defined above with Φ being $r = 3$, the terminating input is the set of all naturals. Among them, those falling within the range $n > 5$ satisfy positive *WPC*, and those belonging to $0 \leq n \leq 5$ satisfy negative *WPC*.

On the other hand, the derived pre-condition pair of $g1$, $(pc^a.g1.\Phi$, $pc^a.g1.\neg \Phi)$, *does not* fully characterize the program inputs with respect to Φ: terminating inputs belonging to $n > 2$ are not captured by any of the derived pre-conditions. Thus, it may not be correct to specialize $g1$ into a perfect form.

Note that full input characterization does not only depend on the contextual constraint, $Ctx^a(g1)$, but also depends on the output constraint. If we choose a new output constraint Φ' to be $r = 1$, then we will have:

$$\begin{array}{lll} pc^a.g1.\Phi' & = [n] : n \leq 2 & = wpc.g1.\Phi' \\ pc^a.g1.\neg\,\Phi' & = [n] : n \leq -1 \vee n \geq 3 & = wpc.g1.\neg\,\Phi' \end{array}$$

Thus, the derived pre-condition pair can fully characterize the program input with respect to Φ'.

Furthermore, derived pre-conditions need not be exactly the same as the *WPC*'s in order to fully characterize the program inputs. Consider the following function definitions headed by $g2$:

$$\begin{array}{lll} g2\ n & = & \mathbf{if}\ n < 3\ \mathbf{then}\ 1 \\ & & \mathbf{else\ if}\ n < 5\ \mathbf{then}\ 2\ \ \mathbf{else}\ h\ n \\ h\ n & = & \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ h\ (n * n) \end{array}$$

Here, $g2$ enters an infinite loop when $n \geq 5$, because function h does not terminate when $n \neq 0$. In the sized type system, infinite recursion is theoretically denoted by *False*. Thus, assuming that the program inputs are naturals, the theoretical contextual constraint is:

$$Ctx^t(g1) = (0 \leq n < 3 \wedge r = 1) \vee (3 \leq n < 5 \wedge r = 2)$$

In practice, however, this size information (for infinite recursion) can be approximated by any Presburger formula. In particular, it is likely for the size system to infer (safely) that $r = 1$ for all calls to h. Thus, the computed contextual constraint can be:

$$\begin{array}{lll} Ctx^a(g1) & = & (0 \leq n < 3 \wedge r = 1) \vee (3 \leq n < 5 \wedge r = 2) \\ & & \vee\ (n \geq 5 \wedge r = 1) \end{array}$$

Now, let the output constraint Φ be $r = 1$, the theoretical positive and negative *WPC*'s for $g2$ with respect to Φ are:

$$\begin{array}{lll} wpc.g2.\Phi & = & [n] : n < 3 \vee n \geq 5 \\ wpc.g2.\neg\,\Phi & = & [n] : n \leq -1 \vee 3 \leq n < 5 \vee n \geq 5 \end{array}$$

whereas the derived pre-conditions are:

$$\begin{array}{lll} pc^a.g2.\Phi & = & [n] : n < 3 \vee n \geq 5 \\ pc^a.g2.\neg\,\Phi & = & [n] : n \leq -1 \vee 3 \leq n < 5 \end{array}$$

Here, the negative pre-condition differs from the negative *WPC*. However, they differ in the set of inputs that do not lead to termination

($n \geq 5$). Hence, the negative pre-condition is a negative *WPC*-variant. The derived pair still correctly partitions the terminating inputs; it thus fully characterizes inputs of $g2$ with respect to Φ.

5.4. Constrained Pre-conditions

In the previous section, we saw how derived pre-conditions can be used to characterize (terminating) inputs of a program with respect to an output constraint. In this section, we provide a sufficient condition for a pair of derived pre-conditions to *fully* characterize the inputs. This will turn the detection of full input characterization into an effective procedure.

Our first attempt to detect full characterization was to rely on the clean partition of terminating inputs by the pair of positive and negative pre-conditions. To do this, we require that all terminating inputs satisfy either positive or negative pre-conditions. A sub-problem to be resolved is thus the detection of *all* terminating inputs, which is known to be undecidable.

Our second attempt was to ignore the terminating-input set, and check for the "disjointness" of positive and negative pre-conditions. By disjointness, we mean that there is no input value that satisfies *both* positive and negative pre-conditions. However, disjointness is not strong enough for detecting full characterization. Even the pair of positive and negative *WPC*'s may not be disjoint, as evidenced in the case for functions $g1$ and $g2$.

Our final, and successful attempt is to check for the "complete coverage" of input domain by positive and negative pre-conditions. To this end, we choose a constrained form of pre-conditions as our derived pre-conditions: We restrict the pre-conditions to capture only those values belonging to the input domain. We call them *constrained pre-conditions*. They are defined as follows:

DEFINITION 4 (Constrained Pre-condition (cpc)). *Given a program P and an output constraint Φ. Let $\mathcal{I}$ be a constraint defining all inputs to P. The* positive *and* negative constrained pre-condition *of P with respect to Φ is defined by:*

$$cpc_{+}^{a} \stackrel{def}{=} (\forall\, X \,.\, (Ctx^{a}(P) \;\Rightarrow\; \Phi)) \;\wedge\; \mathcal{I}$$
$$cpc_{-}^{a} \stackrel{def}{=} (\forall\, X \,.\, (Ctx^{a}(P) \;\Rightarrow\; \neg\, \Phi)) \;\wedge\; \mathcal{I}$$

where X contains all free variables in the formula, except the input size variables.

Analogously, we can define the positive and negative constrained weakest pre-conditions in terms of their original counterparts.

The following table shows the corresponding *cpc*'s of the functions we have mentioned earlier:

	$g1$		$g2$
$\mathcal{I}$	$n \geq 0$		
Φ	$r = 3$	$r = 1$	$r = 1$
pc^a_+	$n \leq -1$	$n \leq 2$	$n < 3 \vee n \geq 5$
pc^a_-	$n \leq 2$	$n \leq -1 \vee n \geq 3$	$n \leq -1 \vee 3 \leq n < 5$
cpc^a_+	*False*	$0 \leq n \leq 2$	$0 \leq n < 3 \vee n \geq 5$
cpc^a_-	$0 \leq n \leq 2$	$n \geq 3$	$3 \leq n < 5$

From the table, we can check that the *cpc*-pair for both $g1$ and $g2$ with respect to the output constraint $r = 1$ "covers" the entire input domain $\mathcal{I}$, whereas the pair for $g1$ with respect to the output constraint $r = 3$ does not. The Following theorem formalizes this fact:[3]

THEOREM 3. **(Full Input Characterization with *cpc*)** *Given a program P and its output constraint Φ , let $\mathcal{I}$ be the constraint defining P's inputs. If $(cpc^a_+ \vee cpc^a_-)$ is equivalent to $\mathcal{I}$, then $(cpc^a_+ \vee cpc^a_-)$ fully characterizes the inputs of P with respect to Φ.*

Proof To show that (cpc^a_+, cpc^a_-) fully characterizes the inputs of P, we need to show that the pair satisfies the three conditions of full characterization on the terminating inputs.

For any terminating input i of P, let r be the result obtained by executing P with input i.

1. i satisfies $\mathcal{I} \Rightarrow i$ satisfies $(cpc^a_+ \vee cpc^a_-)$, by the supposition that $(cpc^a_+ \vee cpc^a_-)$ and $\mathcal{I}$ are equivalent.

2.

	i satisfies cpc^a_+	
$\Leftrightarrow$	i satisfies $(\forall X\,.\,(Ctx^a(P) \Rightarrow \Phi) \wedge \mathcal{I})$	(Definition of cpc^a_+)
$\Rightarrow$	i satisfies $\forall X\,.\,(Ctx^a(P) \Rightarrow \Phi)$	($\wedge$-elimination)
$\Rightarrow$	i satisfies $\forall X\,.\,(Ctx^t(P) \Rightarrow \Phi)$	$(Ctx^t(P) \Rightarrow Ctx^a(P))$
$\Leftrightarrow$	i satisfies $wpc.P.\Phi$	
$\Rightarrow$	r satisfies Φ	(Property 2)

[3] This definition of *cpc* is more general than the concept of "contextualized WPC" in our previous paper [16], in the sense that the former no longer requires the assumption that the input domain $\mathcal{I}$ be equivalent to the context $(\exists X.Ctx^a(P))$. Consequently, the condition for full characterization can be met by larger classes of *cpc*-pairs.

3. Similarly, we can show that i satisfies cpc^a_- implies that r satisfies $\neg\,\Phi$ through the definition of $wpc.P.\neg\,\Phi$. □

6. Specialization

In this section, we present an overview of a simple output-constraint specialization strategy, in which specialization decisions made are based on the result of the analysis described in Section 5. Our objective is to raise and discuss some of the technical issues pertaining to this new form of specialization. We restrict the output-constraint formula to *involve only output (size variables).* Work on developing OCS for programs with respect to constraints in terms of both input and output size variables is currently being investigated.

Furthermore, for ease of presentation, we only consider *monovariant output constraints*. That is, we don't attempt to compute a new output constraint for a sub-expression from the original output constraint for the enclosing sub-expression. For instance, if the following expression has the output constraint $(r\ >\ 3)$:

$$2\ +\ (g\ (n-1))$$

Then an OCS dealing with polyvariant output constraints will need to be aware of the fact that performing OCS recursively on the sub-expression $(g\ (n-1))$ requires a new output constraint $(r\ > 1)$. Using monovariant output constraints simplifies the solution at hand, but reduces the power of OCS to only specializing tail-recursive functions effectively. Correspondingly, the specializer will expect from the analysis phase a set of constrained pre-condition pairs, one for each of the functions, with respect to this output constraint.

Lastly, we assume that the program has been subjected to alpha-conversion, and all variables are given unique names. This again simplifies our presentation.

The entire specialization process involves two specializers: our output-constraint specializer and a conventional constraint-based partial evaluator. For brevity, throughout the section, we refer to our output-constraint specializer as "the specializer".

Recall that a specialized program takes one of the two forms, depending on the ability of its positive and negative cpc^a's to fully characterize the program's inputs.

In the case when $(cpc^a_+,\ cpc^a_-)$ fully characterizes the inputs with respect to an output constraint, the specialized code adopts the perfect form. Here, aggressive specialization is only performed on the conditional branch the evaluation of which leads to a desired output. For this case, it suffices to invoke a traditional constraint-based partial evaluation to transform the original program with respect to cpc_+^a. We refer the readers to the work by Lafave *et al.* [17] for operational detail of such a constraint-based partial evaluator. For brevity, we always refer to this transformation as "*aggressive specialization*".

When (cpc_+^a, cpc_-^a) is not able to fully characterize the inputs, the specialized code adopts the no-test form. Here, the specializer needs to carefully select a group of branches (which always lead to some desired outputs) for aggressive specialization, mark another group of branches (which always lead to undesired output) as error, and wrap another group of branches (the output destination of which is uncertain) with an output-constraint test. For the remainder of this section, we shall focus on defining such an OCS.

6.1. Overview – OCS Decisions

In OCS, the specializer has to make decision about which of the following three tasks to perform:

- Specializing a sub-expression aggressively;
- Replacing a sub-expression by an Error; and
- Wrapping a sub-expression with an output-constraint test.

We call this decision *the OCS decision*.

The first two tasks are performed when the specializer traverses down an expression; the last task is performed while it is on its way back up the modified expression.

During downward traversal, the specializer makes decisions only before entering a branch of an **if**-expression. In addition to output constraints, the specializer carries along a contextual constraint δ. By comparing the contextual constraint against both cpc^a's, the specializer decides to aggressively specialize the branch when the condition δ is stronger than cpc_+^a; it decides to generate error message when δ is stronger than cpc_-^a. In both cases, the specializer returns not only the transformed code, but also a binary marker of value *True*.

If δ is found to be incomparable with both cpc_+^a and cpc_-^a, there are two possible cases: if the specializer is already at the bottom of an expression tree, it simply returns the branch unchanged, *and* a binary marker of value *False*; otherwise, it continues to traverse down the branch, keeping in mind the possible need to wrap the branch with an output-constraint test. As it traverses further down the branch, the specializer has to update the current contextual constraint δ.

Any branch that returns a marker of value *True* indicates that it needs not be wrapped further; a branch which returns a marker of value *False* indicates that it may need to be wrapped with output-constraint test, either at this branch or at some embedding expression.

As an example, consider specializing a variant of function $g1$, called $g1'$, with output constraint $r = 3$. The code of $g1'$ is as follows:

$$\begin{array}{lcl} g1'\ n & = & \textbf{if}\ \ n \leq 2\ \ \textbf{then}\ 1 \\ & & \textbf{else if}\ \ n * n \leq 25\ \ \textbf{then}\ \ n\ \ \textbf{else}\ \ 3 \end{array}$$

The corresponding constrained pre-conditions are: cpc^a_+ is $n = 3$ and cpc^a_- is $0 \leq n \leq 2$. During specialization, traversing the right-hand side of $g1'$ downward yields the following pseudo-code:

$$\begin{array}{l} \textbf{if}\ \ n \leq 2\ \ \textbf{then}\ \ < \mathit{Error}, \mathit{True} > \\ \textbf{else if}\ \ n * n \leq 25\ \ \textbf{then}\ \ < n, \mathit{False} >\ \ \textbf{else}\ \ < 3, \mathit{True} > \end{array}$$

The code has three branches (one **then** branch, and two other branches in the top **else** branch). The first branch is replaced by *Error* because the specializer has determined so from the context of the branch. The second branch is marked with *False* to indicate that wrapping is needed. The third branch returns 3 because the constant result is found to satisfy the output constraint.

On traversing up the expression tree, the specializer has to consider wrapping the branches with an output-constraint test. It does so by examining the markers collected from every branch of an **if**-expression:

1. If one of the branches returns a marker of value *True*, the specializer will wrap all those branches having marker value *False* with an output-constraint test.[4] It then returns the transformed **if**-expression up the expression tree, together with a marker of value *True*. This indicates that it need not be wrapped anymore.

2. If all the branches return markers of value *False*, the specializer simply returns the **if**-expression as it is, together with a marker of value *False*.

Following the example of specializing $g1'$, the specializer will wrap the second branch because its alternate branch (the third branch) has returned a marker of value *True*. The resulting specialized code is as follows:

[4] We use the word "branches" in our description because this technique can be extended to conditional expressions with multiple branches.

$$\begin{array}{l} \textbf{if } n \leq 2 \textbf{ then } \textit{Error} \\ \textbf{else if } n * n \leq 25 \textbf{ then} \\ \qquad\qquad \textbf{let } x = n \textbf{ in if } x = 3 \textbf{ then } x \textbf{ else } \textit{Error} \\ \quad\ \textbf{else } 3 \end{array}$$

At the root of the expression tree, if the marker received from below is *False*, the entire expression is wrapped with an output-constraint test; otherwise, the return expression has already been fully specialized.

6.2. Decision Paths

Our specializer makes OCS decisions only along those paths (in an abstract syntax tree) that lead to a *last* executable sub-expression. We call these paths the *decision paths*. For example, in the following example,

$$\textbf{let } x = e_1 \textbf{ in let } y = e_2 \textbf{ in } x + y$$

the last executable sub-expression is the nested body $x + y$. The decision path consists of two **let**-structures leading to $x + y$. Both e_1 and e_2 are not on the decision path, because they do not contain the last executable expression. As such, they are only subject to constraint-based specialization, not output-constraint specialization.

Through in-lining, expression e_2, and possibly e_1 can be brought into the decision path, and be treated by OCS. As it is, specialization will be more effective only when we allow polyvariant output constraints (*eg.* computing new output-constraints for e_2 in the expression $x + e_2$ after in-lining.)

6.3. Dealing with Function Calls

The decision to unfold or specialize a function call in OCS is similar to that in the context of constraint-based partial evaluation. The result has already appeared in the related literature, notably the work by Lafave and Gallagher in [17, 18, 19]. As such, in this paper, we do not address issues pertaining to infinite unfolding or specialization. We make the assumption that such decision have already been made for each functions.

Treatment of call unfolding is the same as conventional constraint-based partial evaluation. On the other hand, handling call specialization is more involved, and we describe this in detail here.

The specializer will make OCS decisions in the body of a specialized function if the original function call falls on the decision path. In fact, because of monovariant output constraint requirement, the call should be tail-recursive in order to be included in the decision path.

Since we restrict the output constraint to contain only output (size) variables, the output constraint will not be changed when the specialization process is shifted from the caller to the callee. However, the pair of positive and negative constrained pre-condition will be changed from that of the caller to the callee. This is reasonable, because some of the actual arguments might get masked off during the caller/callee shift. Similarly, the contextual constraint at the call site will need to be consolidated, before it can be used as a contextual constraint for the callee. Consolidation of contextual constraint includes gathering all information about variables involved in forming the call argument, and eliminating all (size) variables, through existential quantification, that will become non-local at the body of the callee.

Lastly, we describe the content of the specialization store, called *cache* (of type **Cache**). A cache associates a function name to a pair containing: (1) the *cpc*-pair of the function with respect to the monovariant output constraint, and (2) a list of its specialized functions. Information about a specialized function that is kept in the cache includes *cache information* and a piece of residual code. A cache information item contains the following information, in this order: a specialized function name, a list of parameters, a list of constraints about each of the parameters, and a contextual constraint in which the specialized function has been created. For convenience, information about the original function definition is kept together with the list of its specialized counterparts, and is placed at the end of the list.

There are four operations on a cache: The first operation is to treat the cache as a function, and get a cache entry through function application. Function *cacheIn* puts information of a new specialized function into the cache. *cacheUpd* updates a specialized function's information. *inCache* checks the availability of a specialized function, and returns its name if found.

6.4. The Algorithm

The entire OCS comprises three main functions: a perfect-form transformer $\wp$, an output-constraint specializer (which is defined by a pair of mutually recursive functions $\mathcal{U}$ and $\mathcal{U}'$), and a conventional constraint-based partial evaluator $\mathcal{T}$. In this paper, we omit the definition of $\mathcal{T}$, as its construction can be found in the relevant literature. Figure 4 shows the algorithm for $\wp$, and Figures 7, 8, and 9 show the algorithms for the pair $\mathcal{U}$ and $\mathcal{U}'$.

OCS begins by calling $\wp$ to work on the main function with the following information: an output constraint ϕ, an initial contextual constraint (possibly of value *True*) δ, the main function's input domain $\mathcal{I}$, and an initial cache ϖ_{init} which contains information about all original functions.

The objective of $\wp$ is to transform the main function into a perfect form, if possible. The flow of function $\wp$ is described in Fig. 5. When the *cpc*-pair (Ψ^+, Ψ^-), which can be obtained from the cache, is able to fully characterize program inputs with respect to ϕ, $\wp$ calls the constraint-based partial evaluator $\mathcal{T}$ to aggressively specialize the program with respect to the positive pre-condition Ψ^+ (and the existing context δ). Otherwise, it calls the output-constraint specializer to work on the main function body. Several newly introduced auxiliary functions used by $\wp$ are:

- Function *sizetype* takes an expression, and returns the annotated type and size constraint of the expression.
- Function $\mathcal{L}$ translates a constraint to a boolean-valued expression, so that the latter can be inserted into the specialized program.

Function $\mathcal{U}$ is called to make an OCS decision at a branch. It takes in a function name f as a subscript, an expression e to be specialized, the output constraint ϕ, a contextual constraint δ, a program-variable environment Γ, and the global cache ϖ.

Availability of the function name enables $\mathcal{U}$ to obtain the associated *cpc*-pair for testing. The program-variable environment maps program

$$
\begin{array}{l}
\wp :: \mathbf{Decl} \to \mathbf{F} \to \mathbf{F} \to \mathbf{F} \to \mathbf{Cache} \to \mathbf{Cache} \qquad \textit{where} \\
\quad \begin{array}{lcll}
\Gamma & \in & \mathbf{Env} & = \ \mathbf{Var} \to \mathbf{Exp} \times \mathbf{AnnType} \times \mathbf{F} \\
\varpi & \in & \mathbf{Cache} & = \ \mathbf{Fn} \to ((\mathbf{F} \times \mathbf{F}) \times [\,\mathbf{CInfo} \times \mathbf{Exp}]) \\
\iota & \in & \mathbf{CInfo} & = \ \mathbf{Fn} \times [\,\mathbf{Var}] \times [\,\mathbf{F}] \times \mathbf{F}
\end{array} \\
\wp [\![f \ (x_1, x_2, \ldots, x_n) \ = \ e]\!] \, \phi \, \delta \, \mathcal{I} \, \varpi_{init} \ = \\
\quad \textit{let } \begin{array}[t]{ll}
(\tau_i, \ \psi_i) & = \ \textit{sizetype}[\![\, x_i \,]\!] \\
((\Psi^+, \ \Psi^-), \ \_ \,) & = \ \varpi \, ([\![f]\!]) \\
[\![f']\!] & = \ \textit{newVar} \\
\Gamma & = \ \Gamma_{init}[([\![\, x_i \,]\!], \ \tau_i, \ \psi_i)/x_i] \\
\iota & = \ ([\![f']\!], [[\![x_1]\!], \ \ldots, \ [\![x_n]\!]], \ [\psi_1, \ldots, \psi_n], \delta) \\
\varpi & = \ \textit{cacheIn} \ \ \varpi_{init} \ \ ([\![f]\!], \ \iota) \\
[\![e_{\Psi+}]\!] & = \ \mathcal{L}[\![\Psi^+]\!] \ \Gamma
\end{array} \\
\quad \textit{in if } (\Psi^+ \ \vee \ \Psi^- \ = \mathcal{I}\,) \\
\qquad \textit{then let } \ ([\![e']\!], \ \varpi') \ = \ \mathcal{T} [\![e]\!] \ (\delta \ \wedge \ \Psi^+) \ \Gamma \, \varpi \\
\qquad\qquad \textit{in cacheUpd } \ \varpi' \ \ ([\![f]\!], \ \iota, \ [\![\, \mathbf{if} \ e_{\Psi+} \ \mathbf{then} \ e' \ \mathbf{else} \ \textit{Error} \,]\!]) \\
\qquad \textit{else let } \ ([\![e']\!], \varpi', \ w) \ = \ \mathcal{U}'_f [\![e]\!] \, \phi \, \delta \, (\Psi^+, \Psi^-) \ \Gamma \, \varpi \\
\qquad\qquad\qquad [\![e'']\!] \ = \ \textit{if } w \textit{ then } [\![e']\!] \\
\qquad\qquad\qquad\qquad\qquad \textit{else let } \ [\![e_\phi]\!] = \mathcal{L}[\![\phi]\!] \ \ \Gamma[([\![x]\!], \mathbf{Int}^r, \textit{True})/x] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{in } [\![\, \mathbf{let} \ x = e' \ \mathbf{in} \ \mathbf{if} \ e_\phi \ \mathbf{then} \ x \ \mathbf{else} \ \textit{Error}]\!] \\
\qquad\qquad \textit{in cacheUpd } \varpi' \, ([\![f]\!], \ \ \iota, \ [\![e'']\!])
\end{array}
$$

Figure 4. Specialization Rule – $\wp$

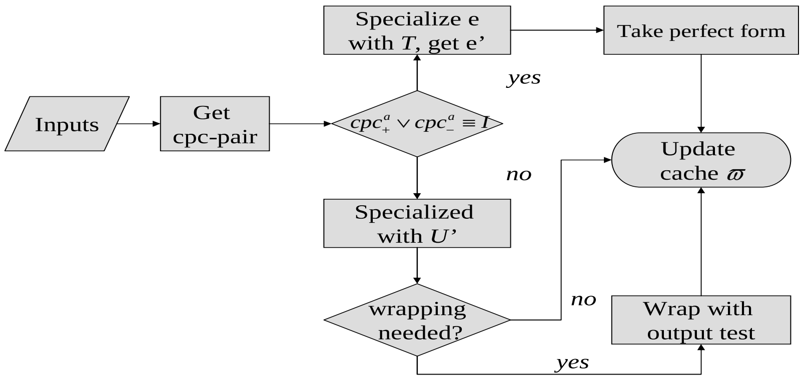

Figure 5. The Process of Function $\wp$

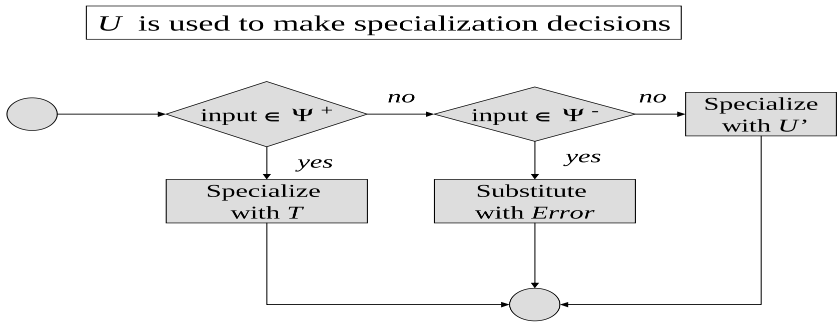

Figure 6. The Process of Function $\mathcal{U}$

variables to a triple consisting of an expression (which is assumed to have been specialized by the constraint-based partial evaluator, but not the output-constraint specialization) , its annotated type, and its size constraint.

The decision process performed by $\mathcal{U}$ has already been discussed in Section 6.1 and is illustrated in Fig. 6. As expected, this is called by function $\mathcal{U}'$ while working on an **if**-expression.

An auxiliary operation is employed by $\mathcal{U}$ to compute the existing context is $\mathcal{F}_{\Gamma,\delta}$. This is a recursively-defined operation that combines (via conjunction) all constraints in Γ which are related directly or indirectly with δ. It produces the best information known about the

$$
\begin{array}{l}
\mathcal{U}_{\_},\, \mathcal{U}'_{\_} \;::\; \mathbf{Fn} \to \mathbf{Exp} \to \mathbf{F} \to \mathbf{F} \to \mathbf{Env} \to \\
\qquad\qquad\qquad \mathbf{Cache} \to (\,\mathbf{Exp} \times \mathbf{Cache} \times \mathbf{Bool})\\
\mathcal{U}_f\, [\![\, e]\!]\;\phi\;\delta\;\Gamma\;\;\varpi \;=\\
\quad \mathit{let}\;\; ((\Psi^+,\; \Psi^-),\; \_\,) \;=\; \varpi\,[\![\, f\,]\!]\\
\quad \mathit{in}\;\; \mathit{if}\;\; (\overline{\exists\, I}\,.\, \mathcal{F}_{\Gamma,\delta} \Rightarrow\; \Psi^+)\\
\qquad \mathit{then}\;\; \mathit{let}\;\; ([\![\, e'\,]\!],\; \varpi') \;=\; \mathcal{T}\,[\![\, e\,]\!]\;\delta\;\Gamma\;\varpi\;\mathit{in}\;\; ([\![\, e'\,]\!],\; \varpi',\; \mathit{True})\\
\qquad \mathit{else}\;\; \mathit{if}\;\; (\overline{\exists\, I}.\, \mathcal{F}_{\Gamma,\delta} \Rightarrow\; \Psi^-)\;\mathit{then}\;([\![\, \mathit{Error}\,]\!],\; \varpi,\; \mathit{True})\\
\qquad\qquad\qquad\qquad\qquad \mathit{else}\; \mathcal{U}'_f\, [\![\, e\,]\!]\;\phi\;\delta\;\Gamma\;\;\varpi
\end{array}
$$

$$
\begin{array}{l}
\mathcal{U}'_f\, [\![\, c\,]\!]\;\phi\;\delta\;\Gamma\;\;\varpi \;\;=\\
\quad \mathit{if}\;\; (r = c) \;\Rightarrow\; \phi\;\;\mathit{then}\;([\![\, c\,]\!],\; \varpi,\; \mathit{True})\\
\qquad\qquad\qquad\qquad\;\; \mathit{else}\;([\![\, \mathit{Error}\,]\!], \varpi,\; \mathit{True})\\
\mathcal{U}'_f\, [\![\, x\,]\!]\;\phi\;\delta\;\Gamma\;\;\varpi \;=\\
\quad \mathit{let}\;\; ([\![\, e\,]\!],\; \_,\; \_\,) \;=\; \Gamma\;\;[\![\, x\,]\!]\\
\quad \mathit{in}\;\;\; \mathit{case}\, [\![\, e\,]\!]\;\mathit{of}\\
\qquad [\![\, c\,]\!] \qquad\qquad \to\; \mathcal{U}'_f\, [\![\, c\,]\!]\;\phi\;\delta\;\Gamma\;\;\varpi\\
\qquad \_ \qquad\qquad\quad\;\; \to\; ([\![\, e]\!],\; \varpi,\; \mathit{False})\\
\mathcal{U}'_f\, [\![\, \mathit{prim}_{op}\;(x_1, \ldots, x_n)]\!]\;\phi\;\delta\;\Gamma\;\;\varpi =\\
\quad \mathit{let}\;\; ([\![\, e\,]\!],\; \varpi') \;=\; \mathcal{T}\,[\![\, \mathit{prim}_{op}\;(x_1, \ldots, x_n)\,]\!]\;\;\delta\;\;\Gamma\,\varpi\\
\quad \mathit{in}\;\; \mathit{case}\, [\![\, e\,]\!]\;\mathit{of}\\
\qquad [\![\, c\,]\!]\;\; \to\; \mathcal{U}'_f\, [\![\, c\,]\!]\;\phi\;\delta\;\Gamma\;\;\varpi'\\
\qquad \_ \qquad\;\; \to\; ([\![\, e]\!],\; \varpi',\; \mathit{False})
\end{array}
$$

For ease of presentation, we assume that program output be of annotated type $\mathbf{Int}^r$. I is the set containing all input size variables. We write $\overline{\exists X}.\,\phi$ as a shorthand for $\exists\, Y.\,\phi$, where $Y\; = fv(\phi)\; -\; X$.

Figure 7. Specialization Rules – $\mathcal{U}$ and $\mathcal{U}'$ (Part I)

current contextual constraint of an expression. Formally, this is defined as follows:

$$
\begin{array}{lcl}
\mathcal{F}_{\Gamma,\delta} & = & \wedge\,(\cup_{i\,\geq\,0}\;\Phi_i) \qquad \textbf{where}\\
\quad \Phi_0 & = & \{\delta\}\\
\quad \Phi_{i+1} & = & \{\;\phi\;\mid\;(\exists\, x,\;\tau\,.\,\Gamma\,[\![\, x\,]\!] \;=\; (\tau, \phi))\,\wedge\\
 & & \qquad\qquad (fv(\phi)\;\cap\; fv(\Phi_i)\;\neq\;\emptyset)\;\wedge\;(\phi\;\notin\;\Phi_i)\;\}
\end{array}
$$

As the environment Γ is finite, computation of $\mathcal{F}_{\Gamma,\phi}$ always terminates.

Function $\mathcal{U}'$ operates on the syntactic constructs of **Exp**. It submits those sub-expressions not in any decision paths to the constraint-based partial evaluator $\mathcal{T}$ for specialization.

Upon encountering a constant in the decision path, $\mathcal{U}'$ checks the constant for output-constraint satisfiability, and returns the appropriate transformed code (either a constant or an error) with a marker of value *True*, indicating that no further wrapping is needed on the returned code.

For variable constructs, $\mathcal{U}'$ retrieves the relevant expression from the environments, and subjects it to a constant check against the output-constraint.

$$
\begin{array}{l}
\mathcal{U}'_f\,[\![\,\mathbf{if}\ \ e_0\ \ \mathbf{then}\ \ e_1\ \ \mathbf{else}\ \ e_2\,]\!]\ \ \phi\ \delta\ \ \Gamma\ \ \ \varpi\ \ = \\
\quad \mathit{let}\ \ (\mathbf{Bool}^v,\ \psi)\ \ =\ \ \mathit{sizetype}\,[\![\,e_0\,]\!] \\
\quad \qquad V \qquad\qquad =\ \ \{v\} \\
\quad \qquad \Phi \qquad\qquad =\ \ \mathcal{F}_{\Gamma,\delta}\ \wedge\ \psi \\
\quad \mathit{in}\ \ \mathit{if}\ \ \overline{\exists\ V}.(\Phi\ \Rightarrow\ (v=1)) \\
\qquad \mathit{then}\ \ \mathcal{U}_f\,[\![\,e_1\,]\!]\ \ (\delta\wedge\psi\wedge(v=1))\ \ \Gamma\ \ \ \varpi \\
\qquad \mathit{else\ if}\ \ \overline{\exists\ V}.(\Phi\ \Rightarrow\ (v=0)) \\
\qquad\quad \mathit{then}\ \ \mathcal{U}_f\,[\![\,e_2\,]\!]\ \ (\delta\wedge\psi\wedge(v=0))\ \ \Gamma\ \ \ \varpi \\
\qquad\quad \mathit{else\ let}\ ([\![\,e'_0\,]\!],\ \varpi_0) \qquad =\ \mathcal{T}\,[\![\,e_0\,]\!]\ \ \delta\ \ \Gamma\ \ \varpi \\
\qquad\qquad\quad ([\![\,e'_1\,]\!],\varpi_1,w_1)\ \ =\ \mathcal{U}_f\,[\![\,e_1\,]\!]\ \ \phi\ \ (\delta\ \wedge\ \psi\ \wedge\ (v\ =\ 1))\ \ \Gamma\ \ \ \varpi_0 \\
\qquad\qquad\quad ([\![\,e'_2\,]\!],\ \varpi_2,w_2)\ \ =\ \mathcal{U}_f\,[\![\,e_2\,]\!]\ \ \phi\ \ (\delta\ \wedge\ \psi\ \wedge\ (v\ =\ 0))\ \ \Gamma\ \ \ \varpi_1 \\
\qquad\qquad \mathit{in}\ \ \mathit{if}\ \ (w_1\ \wedge\ w_2)\ \vee\ \neg(w_1\ \vee\ w_2) \\
\qquad\qquad\quad \mathit{then}\ \ ([\![\,\mathbf{if}\ e'_0\ \mathbf{then}\ e'_1\ \mathbf{else}\ e'_2\,]\!],\ \varpi_2,w_1) \\
\qquad\qquad\quad \mathit{else}\ \ \mathit{let}\ [\![\,e_\phi\,]\!]\ \ =\ \ \mathcal{L}[\![\,\phi_g\,]\!]\ \ \Gamma[([\![\,x\,]\!],\mathbf{Int}^r,\mathit{True})/x] \\
\qquad\qquad\qquad\qquad [\![\,e''_1\,]\!]\ \ =\ \ \mathit{if}\ w_1\ \mathit{then}\ [\![\,e'_1\,]\!]\ \mathit{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![\,\mathbf{let}\ x=e'_1\ \mathbf{in\ if}\ e_\phi\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{Error}]\!] \\
\qquad\qquad\qquad\qquad [\![\,e''_2\,]\!]\ \ =\ \ \mathit{if}\ w_2\ \mathit{then}\ [\![\,e'_2\,]\!]\ \mathit{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![\,\mathbf{let}\ x=e'_2\ \mathbf{in\ if}\ e_\phi\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{Error}]\!] \\
\qquad\qquad\qquad \mathit{in}\ ([\![\,\mathbf{if}\ e'_0\ \mathbf{then}\ e''_1\ \mathbf{else}\ e''_2\,]\!],\ \varpi_2,\ \mathit{True})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{U}'_f\,[\![\,\mathbf{let}\ \ x\ =\ e_1\ \ \mathbf{in}\ \ e_2\,]\!]\ \ \phi\ \delta\ \ \Gamma\ \ \varpi\ \ = \\
\quad \mathit{let}\ \ (\tau^v,\ \psi) \qquad =\ \ \mathit{sizetype}[\![\,e_1\,]\!] \\
\qquad ([\![\,e'_1\,]\!],\ \varpi')\ \ =\ \ \mathcal{T}\,[\![\,e_1\,]\!]\ \delta\ \Gamma\ \varpi \\
\quad \mathit{in}\ \ \mathit{case}\ \ [\![\,e'_1]\!]\ \ \mathit{of} \\
\qquad [\![\,c\,]\!]\ \rightarrow\ \mathcal{U}'_f\,[\![\,e_2\,]\!]\ \ \phi\ \ \delta\ \ \Gamma[([\![\,c\,]\!],\tau,v=c)/x]\ \ \ \varpi' \\
\qquad \_ \qquad \rightarrow\ \ \mathit{let}\ ([\![\,e'_2\,]\!],\varpi'',w)\ \ =\ \ \mathcal{U}'_f\,[\![\,e_2\,]\!]\ \ \phi\ \delta\ \Gamma[([\![\,e'_1\,]\!],\tau,\ \psi)/x]\ \ \ \varpi' \\
\qquad\qquad\quad \mathit{in}\ ([\![\,\mathbf{let}\ \ x\ =\ e'_1\ \ \mathbf{in}\ \ e'_2\,]\!],\ \varpi'',\ w)
\end{array}
$$

Figure 8. Specialization Rules – $\mathcal{U}'$ (Part II)

$\mathcal{U}'$ submits a primitive operation to $\mathcal{T}$ for specialization, before performing a constant check on the result.

For the **if**-expression, $\mathcal{U}'$ calls its counterpart $\mathcal{U}$ to make the OCS decision. Notice that the contextual constraint δ is updated with information from the **if**-test, before being passed to $\mathcal{U}$. Upon returning from calls to $\mathcal{U}, \mathcal{U}'$ has to decide if it need to wrap the branches with an output-constraint test, based on the markers returned at each branch. Only when $(w_1 \neq w_2)$, the branch returning $False$ will be wrapped up. This process is illustrated in Fig. 10.
For the **let**-expression, $\mathcal{U}'$ sends the local abstract to $\mathcal{T}$ for specialization, updates the environment with local information, and recursively calls itself to work on the **let**-body. The process is illustrated in Fig. 11.

For a call to a user-defined function, say g, we assume that the unfold/specialize decision has been provided by the user (or some decision procedure outside OCS), and can be accessed via the *unfold*? call.

$$
\begin{array}{l}
\mathcal{U}'_f [\![\, g \ (x_1, \ldots, x_n) \,]\!] \ \ \phi \ \delta \ \ \Gamma \ \ \ \varpi \ = \\
\quad \mathit{let} \ \ (\_, \ \mathit{clist}) \qquad\qquad\quad = \ \varpi \, [\![\, g \,]\!] \\
\qquad\quad ((\_ \, , \mathit{ys}, \_ \, , \_ \,), \ e_g) \ \ = \ \alpha \, (\mathit{last} \ \ \mathit{clist}) \\
\qquad\quad [[\![\, y_1 \,]\!], \ldots, [\![\, y_n \,]\!]] \quad = \ \mathit{ys} \\
\qquad\quad ([\![\, e_i]\!] \, , \ \tau_i, \ \psi_i) \ \ = \ \Gamma \ x_i \ \ \forall \, i \ = \ 1 \ \ldots \ n \\
\qquad\quad X \qquad\qquad\qquad\quad = \ \bigcup_{i=1}^{n} \, \{ \mathit{fv}(\tau_i) \} \\
\qquad\quad \mathcal{F}' \qquad\qquad\qquad\ = \ \overline{\exists \, X} . \ \mathcal{F}_{\Gamma, \delta} \\
\qquad\quad [\![\, g' \,]\!] \qquad\qquad\quad\ = \ \mathit{newVar} \\
\qquad\quad \iota \qquad\qquad\qquad\quad\ \ = \ ([\![\, g' \,]\!], \mathit{ys}, \ [\psi_1, \ldots, \psi_n], \ \mathcal{F}') \\
\quad \mathit{in} \ \mathit{if} \ (\mathit{unfold?}(g)) \ \mathit{then} \ \ \mathcal{U}'_f \, [\![\, e_g \,]\!][x_i / y_i] \ \ \phi \ \delta \ \ \Gamma \ \ \ \varpi \\
\qquad \mathit{else} \ \ \mathit{case} \ \ (\mathit{inCache} \ \ ([\![\, g \,]\!] \, , \iota)) \ \ \mathit{of} \\
\qquad\quad [\![\, g']\!] \ \ \rightarrow \ \ ([\![\, g'(x_1, \ldots, x_n) \,]\!], \ \varpi, \ \ \mathit{True}) \\
\qquad\quad \_ \qquad \rightarrow \ \mathit{let} \ \ \varpi' \ \ = \ \ \mathit{cacheIn} \ \ \varpi \ \ ([\![\, g \,]\!], \ \iota) \\
\qquad\qquad\qquad\qquad ([\![\, e'_g \,]\!], \varpi'', \ w \ ) \ \ = \ \mathcal{U}_{\ g} \, [\![\, e_g \,]\!] \ \ \phi \ \ \mathcal{F}' \ \ \Gamma_g \ \ \varpi' \\
\qquad\qquad\qquad\qquad \Gamma_g \qquad\ = \ \Gamma_{\mathit{init}} [([\![\, y_i \,]\!], \tau_i, \psi_i) / y_i] \\
\qquad\qquad\qquad\qquad [\![\, e_\phi \,]\!] \ \ = \ \mathcal{L} [\![\, \phi_g \,]\!] \ \ \Gamma_g [([\![\, x \,]\!], \mathbf{Int}^r, \ \mathit{True}) / x] \\
\qquad\qquad\qquad\qquad [\![\, e''_g \,]\!] \ \ = \ \mathit{if} \ w \ \mathit{then} \ [\![\, e'_g \,]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else} \ [\![\, \mathbf{let} \ x = e'_g \ \mathbf{in} \ \mathbf{if} \ e_\phi \ \mathbf{then} \ x \ \mathbf{else} \ \mathit{Error}]\!] \\
\qquad\qquad\qquad\qquad \varpi''' \qquad = \ \mathit{cacheUpd} \ \ \varpi'' \ ([\![\, g \,]\!], \ \iota, \ [\![\, e''_g \,]\!]) \\
\qquad\qquad\qquad\quad \mathit{in} \ \ ([\![\, g'(x_1, \ldots, x_n) \,]\!], \ \varpi''', \ \mathit{True})
\end{array}
$$

Figure 9. Specialization Rules – $\mathcal{U}'$ (Part III)

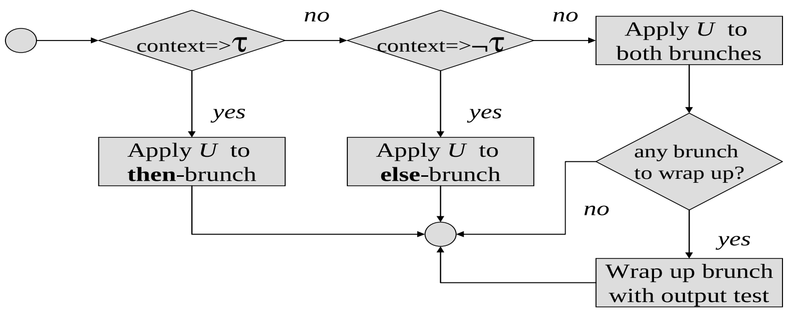

Figure 10. The Process of Function $\mathcal{U}'$ for an **if**-expression

Unfolding a call proceeds just like constraint-based partial evaluation. When a similar specialized function of g cannot be found in the cache during call specialization, a new specialized function is created. Its body is obtained by output-constraint specializing the right-hand side expression of g's definition with respect to the same output constraint, ϕ, *but with the constrained pre-condition pair of* g *with respect to* ϕ. This is reflected in the use of a new function-name parameter in the

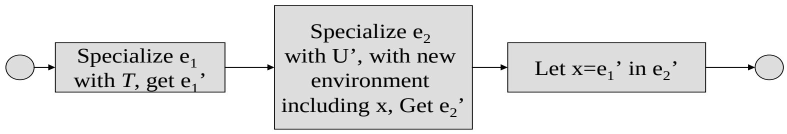

Figure 11. The Process of Function $\mathcal{U}'$ for a **let**-expression

recursive call to $\mathcal{U}'$. Note that we cannot use the existing constrained pre-condition pairs (available at the caller) because information about the actual call arguments, which may contain variables and constraints related to the variables of the caller, may have been lost when the control is moved to the callee. This process is illustrated in Fig. 12. Consider the following tail-recursive function h for summing arguments

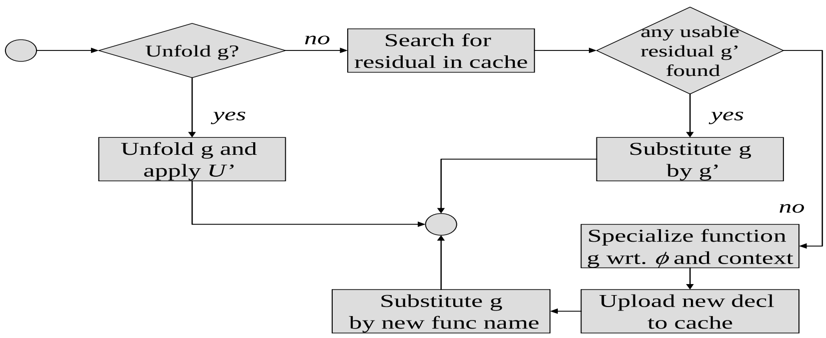

Figure 12. The Process of Function $\mathcal{U}'$ for Function Calls

(assuming both n and m are naturals):

$$
\begin{array}{lcl}
h\ n\ m & = & \textbf{if}\ \ n > 0\ \ \textbf{then}\ h\ (n-1)\ (m+1) \\
 & & \textbf{else}\ m
\end{array}
$$

To illustrate the effect of $\mathcal{U}$ and $\mathcal{U}'$, let us assume that we do not intend to transform the function into perfect form (using $\wp$). Given that the output constraint is $0 < r < 3$, and assuming calls to h are to be unfolded, we obtain the following result:

$$
\begin{array}{lcl}
h1\ n\ m & = & \textbf{if}\ \ n > 0\ \ \textbf{then} \\
 & & \quad \textbf{if}\ \ (n-1) > 0\ \ \textbf{then} \\
 & & \quad\quad \textbf{if}\ \ (n-2) > 0\ \ \textbf{then} \\
 & & \quad\quad\quad \textbf{if}\ \ (n-3) > 0\ \ \textbf{then}\ \mathit{Error}\ \ \textbf{else}\ \mathit{Error} \\
 & & \quad\quad \textbf{else}\ \ \mathit{checkOC}\ (m+2) \\
 & & \quad \textbf{else}\ \ \mathit{checkOC}\ \ (m+1) \\
 & & \textbf{else}\ \mathit{checkOC}\ m
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{checkOC}\ m & = & \mathbf{let}\ x\ =\ m\ \mathbf{in} \\
 & & \mathbf{if}\ (0\ <\ x\ <\ 3)\ \mathbf{then}\ x\ \ \mathbf{else}\ \mathit{Error}
\end{array}
$$

In the above example, we replace any output-constraint test by a call to such a test, in order to avoid cluttering the code.

6.5. Example

A more realistic vending machine simulation is used as an example to illustrate the effectiveness of the OCS approach.

The vending machine *vm* takes in three arguments, the first one *money* means the number of tokens that have been inserted into the machine (initially it is always Zero), the second *bev* refers to the beverage the customer chooses(1 for Coke, 2 for Coffee, and 3 for Tea), the last one *ts* means a series of tokens. Here is the program:

```
vm money  bev  ts  =
   case  ts  of
         []  →  if (bev == 1)
                then  if  (money >= 1)  then  bev  else  0
                else  if  (bev == 2)
                      then  if  (money >= 2)  then  bev  else  0
                      else  if  (bev == 3)
                            then  if  (money >= 3)  then  bev
                                                    else  0
                            else  0
         (t : ts′)  →  vm  (money + 1)  bev  ts′
```

Suppose we only want the vending machine to sell coffee, so the output constraint is $r = 2$. With the initial context $\mathit{money} = 0$, we get the context of the function *vm*.

$$
\begin{array}{l}
\mathit{vm}\ ::\ (\mathbf{Int}^{i}\ \times\ \mathbf{Int}^{j}\ \times\ [\mathbf{Int}]^{k})\ \rightarrow\ \mathbf{Int}^{r} \\
\mathit{Ctx}(\mathit{vm}) = (j = 1) \wedge\ ((k \geq\ 1\ \wedge\ r = 1) \vee (k = 0\ \wedge\ r = 0)) \\
\qquad\qquad\ \vee\ (j = 2)\ \wedge\ ((k \geq\ 2\ \wedge\ r = 2) \vee (0 \leq\ k\ \leq\ 1\ \wedge\ r = 0)) \\
\qquad\qquad\ \vee\ (j = 3)\ \wedge\ ((k \geq\ 3\ \wedge\ r = 2) \vee (0 \leq\ k\ \leq\ 2\ \wedge\ r = 0))
\end{array}
$$

So the constrained pre-condition pairs of *vm* are:

$$
\begin{array}{lcl}
\mathit{cpc}^{a}_{+} & = & (j = 2\ \wedge\ k\ \geq\ 2) \\
\mathit{cpc}^{a}_{-} & = & (j = 1) \vee\ (j = 2)\ \vee\ (j = 2)\ \wedge\ (0 \leq\ k\ \leq\ 1)
\end{array}
$$

The pair "covers" the whole input domain, so it forms a full input characterization. Thus, the specialization can take the perfect form:

$$
\begin{array}{lcl}
\mathit{vm}'\ \mathit{money}\ \ \mathit{bev}\ \ \mathit{ts} & = & \mathbf{if}\ \ (\mathit{length}(\mathit{ts}) \geq\ 2\ \wedge\ \mathit{bev} = 2) \\
 & & \qquad \mathbf{then}\ \ 2\ \ \mathbf{else}\ \ \mathit{Error}
\end{array}
$$

Suppose we cannot get any useful information from the context synthesis, so $Ctx(vm) = True$, and $cpc^a_+ = False$, $cpc^a_- = False$. Clearly, this *cpc*-pair cannot form a full characterization, so we subject the body of vm to the OCS algorithm U' and we get another specialized version of vm:

$$
\begin{array}{l}
vm''\ money\ \ bev\ \ ts\ \ = \\
\quad \textbf{case}\ ts\ \textbf{of} \\
\qquad [\,]\ \rightarrow\ \ \textbf{if}\ (bev = 2) \\
\qquad\qquad\qquad \textbf{then if}\ (money \geq\ 2)\ \textbf{then}\ bev\ \textbf{else}\ Error \\
\qquad\qquad\qquad \textbf{else}\ Error \\
\qquad (t : ts')\ \rightarrow\ \ vm''\ (money + 1)\ \ bev\ \ ts'
\end{array}
$$

After post-processing, the Error branches can be combined together, and we get another version:

$$
\begin{array}{l}
vm_1\ money\ \ bev\ \ ts\ \ = \\
\quad \textbf{case}\ ts\ \textbf{of} \\
\qquad [\,]\ \rightarrow\ \ \textbf{if}\ (bev = 2)\ \wedge\ (money \geq\ 2)\ \textbf{then}\ bev\ \textbf{else}\ Error \\
\qquad (t : ts')\ \rightarrow\ \ vm_1\ \ (money + 1)\ \ bev\ \ ts'
\end{array}
$$

We have only tested OCS on some small programs. We will do more experiments in the future and apply OCS in real world application.

6.6. Discussion

The quality of the specialization result depends on all components involved: the analysis, the constraint-based partial evaluator used ($\mathcal{T}$), and the main specialization functions $\mathcal{U}'$ and $\mathcal{U}$. Here the quality of the analysis result means how accurate the pre-condition is, which depends on how accurate the context capturing process is. The quality of the partial evaluator $\mathcal{T}$ also contributes a lot. By keeping the constraint-based partial evaluator as an independent component within the system, a newly developed partial evaluator can be plugged in to improve the quality of the entire specialization.

Functions $\mathcal{U}$ and $\mathcal{U}'$ as presented have been rather conservative. They can be strengthened in at least the following ways:

1. *Better interaction with function* $\mathcal{T}$: Currently, function $\mathcal{T}$ returns a piece of specialized code. It should also be able to return a constraint describing the size information about the specialized code. As the returned constraint is computed for a specific context during partial evaluation, it will be more precise than the sized type information available at the original expression, which had been collected before output-constraint specialization. With it, the specialized code can be further examined for its satisfiability with the

output constraint. Currently, function $\mathcal{U}'$ only performs a constant check of the specialized code.

Moreover, in the case when the specialized code returned by $\mathcal{T}$ contains an **if**-expression, and it falls in a decision path, we can once again subject the code to the OCS decision process. This happens when there is only a variable in the decision path, and may occur from the result of specializing a primitive operation.

2. *Improving Unfold/Specialize decisions*: Currently, we rely on the user to provide these decisions. While this is fine, the decision is still primitive; it remains at the level of deciding to unfold or specialize a call. A more expressive sub-language should be provided for making such a decision. Many systems (*eg.*: Schism [9]) enable the user to specify how arguments to a function should be treated during call specialization. In OCS, we will also require the user to specify the constraint under which call specialization should take place. For example, in the example of specializing function h, we may wish to annotate the function with information such as: "unfold when $n < 3$."

   For OCS to be useful for component adaptation, a system with automated unfold/specialize decisions is desirable. Further work in this direction is still required.

Furthermore, it is certainly desirable to post-process the specialized code, such as eliminating the test in the case when all branches that have consistent values (*eg. Error*), or to in-line a local definition which appears once in the **let**-body, and many others.

Lastly, the algorithm has been restricted to the specialization of a program with respect to monovariant output-only constraints. A polyvariant output-only constraint can be handled during pre-processing phase by propagating output constraints inwards to the sub-expressions. Handling of the general polyvariant input-output-related constraint can be challenging, as a sub-expression may partially satisfy a constraint, with the rest of the constraints to be satisfied by another sub-expression. The work in this direction is currently in progress.

7. Related Work

The main objective of OCS is to adapt a program to a new form of constraint: the output constraint. As such, the treatment of a program is quite different from the conventional partial evaluation approach [10, 15], which specializes programs with respect to input information.

In the first place, it is not clear if any productive input information can be derived from an output constraint. Next, even when all program inputs are fully characterized with respect to an output constraint, the outcome of OCS – a specialized program – is still expected to accept all possible inputs. This expectation can adversely affect the aggressiveness of specialization, since the latter must ensure that all input are treated correctly. To appropriately relate partial evaluation to OCS, we can say that partial evaluation performs a kind of "positive" specialization, whereas OCS requires specialization of both "positive" and "negative" information. (We note, however, that the term "positive" used here is similar in spirit, but different in practice, from the term "positive supercompilation" used in the partial-evaluation community [23].) Furthermore, partial evaluation assumes the availability of "positive" information, whereas OCS requires derivation of both "positive" and "negative" information.

Another area of research that is closely related to the idea of OCS is *program slicing* [28, 27, 7]. Some recent work in this area has focused on deriving program slices based on post-conditions, such as *p-slicing* [8]. However, P-slicing does not necessarily produce a program slice that is semantically equivalent to the original program, given a particular program input. Other program slicing techniques, noticeably the conditional slicing [3], do not state any requirement for the correctness of program slices in the situation when the input does not satisfy the *WPC* of a program with respect to the post-condition. Therefore, it allows the derivation of a pre-condition that is weaker than *WPC*. Tom Reps *et al.* have done a similar work, applying program slicing on functional programs [22], thus achieving the effect of program specialization. However, they have not looked into expressing the output slicing criteria in terms of constraints, and the resulting program is a program slice, not really an adapted program with optimization in mind. On the other hand, theirs was the first work that looked into specialization with respect to output conditions. We believe that more research is needed to attain a synergy between program slicing and program specialization.

Our work shares similar spirit with the work on inverse computation [1, 24]. Both attempt to find a class of inputs that can lead to an output constraint. While the inverse computation produces an inverse program, we do not reverse the control flow of the original program.

On the analysis aspect, there is abundant work on deriving weakest pre-conditions from a given program output. The basic idea behind the backward derivation of weakest pre-condition was already present in the inductive iteration method, pioneered by Suzuki and Ishihata[25], and more recently improved by Xi *et al.* [30] and Flanagan *et al.*[13].

An earlier version of forward context analysis appeared in [6], with the intention to find total- and partial-redundant checks in a program. A similar version, without introducing sized types, can be found in [26]. In this paper, we rely heavily on contextual information, both to derive weakest pre-conditions, and to characterize program inputs.

8. Conclusions

In this paper we introduce a novel concept of program specialization based on output constraints. We describe how an efficient specialized program should behave, and illuminate an approach to attain this efficiency, while minimizing the number of additional tests required in specialized programs. In the process, we translate output constraints to a characterization function for program's input, and define a specializer that uses this characterization to guide the specialization process. We argue that full characterization of inputs can reduce the number of tests, and provide a sufficient condition for detecting the existence of full characterization.

The theorem of full characterization assumes that derived pre-conditions are stronger than theoretical WPC. This is in the spirit of *must*-analysis. Because we capture both positive and negative pre-conditions We can easily extend this full-characterization theorem to work on "pre-conditions" which are weaker than the theoretical WPC — in the spirit of *may*-analysis.

The specializer presented only serves as a proof of concept. Much work is still required to fine-tune our specializer. In particular, the termination issue of our specialization has not been addressed.

Lastly, our work can be extended in many ways:

1. By combining techniques for input-constraint specialization (*a.ka.* partial evaluation) and output-constraint specialization, we now have better insight into specializing programs with respect to a constraint occurring anywhere in a program. We believe this will broaden the applicability of program specialization, and make the latter a promising tool for program adaptation.

2. Instead of one output constraint, we may wish to assert multiple constraints in a program. These constraints need not be identical. We believe the technique for handling them remains almost the same: searching for a full characterization of program inputs with respect to different combinations of such constraints.

3. The work can be deployed to different paradigms of programming languages. Specifically, it may be interesting to find out the formal

relationship between output-constraint specialization and program slicing with *WPC* [8]. Moreover, we are currently looking into application of this work to languages with imperative features, as well as other programming features such as non-determinism and concurrency.

We believe this work will broaden the scope of program specialization, and provide a framework for building more generic and versatile program adaptation techniques.

9. Acknowledgments

We would like to thank Wei-Ngan Chin and Neil Jones for their critical review of the work, and Corneliu Popeea and Dana Na Xu for their valuable comments on the first draft of the paper. We also thank the anonymous referees for their insightful and detailed comments, and Hugh Anderson for proof-reading the paper. This work has been supported by the research grants R-252-000-092-112 and R-252-000-138-112.

References

1. S. M. Abramov and R. Glück. The universal resolving algorithm: inverse computation in a functional language. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 187–212. Springer-Verlag, 2000.
2. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
3. G. Canfora and A. Cimitile and A. De Lucia. Conditional Program slicing. *Information and Software Technology*, 40:595–607, Elsevier, 1998.
4. W. N. Chin and S. C. Khoo. Calculating sized types. In *2000 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72, Boston, Massachusetts, United States, January 2000.
5. W. N. Chin and S. C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computing (HOSC)*, 14(2/3), 2001.
6. W. N. Chin, S. C. Khoo, and D. X. Xu. Deriving pre-conditions for array bound check elimination. In *Proceedings of the Second Symposium on Programs as Data Objects, PADO 2001*, pages 2–24, Aarhus, Denmark, May 2001.
7. I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *Proceedings of 2001 ACM Symposium on Applied Computing*, pages 605–609, Las Vegas, Nevada, United States, 2001.
8. J.J. Comuzzi and J.M. Hart. Program slicing using weakest preconditions. In *FME '96: Industrial Benefit and Advances in Formal Methods*, volume LNCS 1051, pages 557–575, Oxford, England, March 1996. Springer-Verlag.
9. C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66-77, 1993.

10. C. Consel. Program adaptation based on program transformation. *ACM Computing Survey*, 28 (4es)(164), 1996.
11. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.
12. H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.
13. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Symposium on Principles of Programming Languages*. ACM Press, January 2001.
14. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
15. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, June 1993. xii + 415 pages. ISBN 0-13-020249-5.
16. S. C. Khoo and K. Shi. Output Constraint Specialization. In *ACM SIGPLAN ASIA Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 106–116, Aizu, Japan; 12 - 14 September 2002.
17. L. Lafave and J. P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In *Program Synthesis and Transformation. 7th International Workshop, LOPSTR'97*, pages 168–188, Leuven, Belgium, (LNCS 1463), July 1997.
18. L. Lafave. *A Constraint-based Partial Evaluator for Functional Logic Programs and its Application.* PhD thesis, University of Bristol, 1998.
19. L. Lafave and J. P. Gallagher. Extending the power of automatic constraint-based partial evaluators. *ACM Computing Survey*, 30 (3es), pages 1–4, 1998.
20. G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4), pages 517–561, 1989.
21. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.
22. T. Reps and T. Turnidge. Program specialization via program slicing. *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, Schloss Dagstuhl, Wadern, Germany, 409–429, (LNCS 1110), Feb, 1996, O. Danvy, R. Glueck, and P. Thiemann (eds.)
23. M. Sørensen and R. Glück and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:811-838, 1996.
24. J. P. Secher and M. Sørensen. From checking to inference via driving and dag grammars. In Peter Thiemann, editor, *2002 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 41–51, January 2002.
25. N. Suzuki and K. Ishihata. Implementation of array bound checker. In *ACM Principles of Programming Languages*, pages 132–143, 1977.
26. A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *1991 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11, New Haven, Connecticut, United States, June, 1991.
27. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121-189, September 1995.
28. G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 26(6), pages 107–119. ACM Press, June 1991.
29. H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

30. Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 70–82. ACM Press, June 2000.

Appendix

A. Proof of Theorem 1

Theorem 1 [*WPC* **of Functional Programs**] Given a program P and an assertion Φ about its output. Denote the result of performing $\mathcal{C}$ over P by $Ctx(P)$. Let P' be the corresponding PGCL program translated from P. Then,

$$wpc.P'.\Phi = \forall X.(Ctx(P) \Rightarrow \Phi).$$

where X contains all free variables in the formula, except the input size variables.

Proof

It suffices to prove that for any expression e which constitutes the body (*ie.*, right hand side) of P, if π is the corresponding translated sub-program in PGCL language, then we have

$$wpc.\pi.\Phi = \forall X.(Ctx(e) \Rightarrow \Phi)$$

where X contains all free (size) variables in the formula, except the size variables associated with free variables of e (in other words, these free variables are viewed as inputs to the expression.)

We first translate e to a program π in Passified Guarded Command Language (PGCL) [13]. This gives us the ability to describe the weakest pre-condition of e in terms of the weakest pre-condition of π.

PGCL Overview: PGCL is a variant of Dijkstra's guarded command language [11]. PGCL includes assume and assert statements, sequential composition, demonic (non-deterministic) choice, and function calls. *assume* ϕ act as a "guard", and terminates normally if the predicate ϕ evaluates to *True*, and simply cannot be executed from a state where ϕ evaluates to *False*. The execution of the choice statement $A \,[\,]\, B$ arbitrary chooses either A or B to execute. This non-determinism can be tamed by *assume* statement: Consider the following statement:

$$(assume\ \phi\ ;\ A)\ [\,]\ (assume\ \neg\phi\ ;\ B)$$

It is deterministic because there is only one valid branch to choose for execution.

There is no assignment statement in PGCL. Assignment has been translated into the assume statement by a function call $passify$. The basic idea is to replace each assignment statement

$$x\ :=\ e$$

by an assumption

$$assume\ x'\ =\ e$$

where x' is a fresh variable, and to change subsequent references to x to refer to x'.

Defining wp and ctx: Given a program π in PGCL and an assertion Φ about its output. Define a *context* of π called $ctx(\pi)$ as shown in Table 1. The second column of Table 1 defines the weakest pre-condition semantics of PGCL (this is given by the semantics of PGCL, which was described in [20]). The third column defines the context of π in a syntax-directed manner. Together, the second and third column shows the relationship between the weakest pre-condition of statements in PGCL and the context of the corresponding statements. We first prove that this relation asserts the following properties:

$$wp.\pi.\Phi = (ctx(\pi) \Rightarrow \Phi).$$

Table I. Syntax-directed Rules

Syntax of π	$wp.\pi.Q$	$ctx(\pi)$
$skip$	Q	$True$
$assume\ e$	$e \Rightarrow Q$	e
$A;B$	$wp.A.(wp.B.Q)$	$ctx(A) \wedge ctx(B)$
$A[\!]B$	$wp.A.Q \wedge wp.B.Q$	$ctx(A) \vee ctx(B)$
$p(x,y)$	$ctx(p) \Rightarrow Q$	$ctx(p)$

$$[\pi = assume\ e\ ] \quad \begin{aligned} wp.\pi.\Phi &= e \Rightarrow \Phi \\ &= ctx(\pi) \Rightarrow \Phi \end{aligned}$$

$$[\pi = A;B\ ] \quad \begin{aligned} wp.\pi.\Phi &= wp.A.(wp.B.\Phi) \\ &= ctx(A) \Rightarrow wp.B.\Phi && \text{(structural induction)} \\ &= ctx(A) \Rightarrow (ctx(B) \Rightarrow \Phi) && \text{(structural induction)} \\ &= (ctx(A) \wedge ctx(B)) \Rightarrow \Phi && (p \Rightarrow (q \Rightarrow r) = (p \wedge q) \Rightarrow r) \\ &= ctx(A;B) \Rightarrow \Phi \end{aligned}$$

$$[\pi = A[\!]B\ ] \quad \begin{aligned} wp.\pi.\Phi &= wp.A.\Phi \wedge wp.B.\Phi \\ &= (ctx(A) \Rightarrow \Phi) \wedge (ctx(B) \Rightarrow \Phi) && \text{(structural induction)} \\ &= (ctx(A) \vee ctx(B)) \Rightarrow \Phi && ((p \Rightarrow r) \wedge (q \Rightarrow r) = (p \vee q) \Rightarrow r) \\ &= ctx(A[\!]B) \Rightarrow \Phi \end{aligned}$$

The case for $\pi = skip$ and $\pi = p(x,y)$ is obvious. Thus, by structural induction on the syntax of passified Guarded Command Language(PGCL), we have shown

$$wp.\pi.\Phi = (ctx(\pi) \Rightarrow \Phi)$$

Translating to PGCL: Next, we show in Fig. 13 the rules to transform a program $Prog$ to its PGCL form. These rules are extended from the rules of context computation defined in Figure 3.

The function $\mathcal{D}$ takes in an expression, e, in our language, and returns a triple comprising of a sized type of e, a contextual constraint ϕ, expressed in terms of the size variables, and a PGCL code π composed using size variables. Note that $Ctx(e)$, the result of performing $\mathcal{C}$ over e, is the same as ϕ defined in $\mathcal{D}$. Furthermore, we define $ctx(\pi) = \phi$.

Proof of the theorem: Now, we can link together an expression e in our language and the corresponding program π in PGCL, through the common size

$$
\begin{array}{l}
\mathcal{D}, \mathcal{C}_{main} \;::\; \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{F} \rightarrow (\textbf{AnnType} \times \textbf{F} \times \textbf{PGCL}) \\
\qquad \textit{where} \;\; \textbf{Env} \;=\; \textbf{Var} \rightarrow \textbf{AnnType} \times \textbf{F} \\
\qquad\qquad \textbf{PGCL} \;=\; \textit{PGCL programs} \\
\mathcal{C}_{main} [\![e]\!] \; \Gamma \, \psi \;=\; \textit{let} \; (\tau^{v}, \phi, \pi) \;=\; \mathcal{D} [\![e]\!] \; \Gamma \, \psi \\
\qquad\qquad\qquad\qquad \pi' \;=\; [\![\textit{assume} \; r = v;]\!] \\
\qquad\qquad\qquad \textit{in} \;\; (\_ , \phi \wedge (r = v), \pi; \pi') \\
\mathcal{D} [\![x]\!] \; \Gamma \, \psi \;=\; \textit{let} \; (\tau^{v_1}, \phi) \;=\; \Gamma [\![x]\!] \\
\qquad\qquad\qquad\qquad v \;=\; \textit{newVar} \\
\qquad\qquad\qquad \textit{in} \;\; (\tau^{v}, (v = v_1), [\![\textit{assume} \; v = v_1;]\!]) \\
\mathcal{D} [\![n]\!] \; \Gamma \, \psi \;=\; \textit{let} \; v \;=\; \textit{newVar} \;\; \textit{in} \; (\textbf{Int}^{v}, (v = n), [\![\textit{assume} \; v = n;]\!]) \\
\mathcal{D} [\![f \; (x_1, \ldots, x_n)]\!] \; \Gamma \, \psi \;=\; \\
\qquad \textit{let} \; ((\tau_1{}^{v_1}, \ldots, \tau_n{}^{v_n}) \rightarrow \tau, \phi_f) \;=\; \alpha \, (\Gamma [\![f]\!]) \\
\qquad\qquad Y \;=\; \cup_{i=1}^{n} \{v_i\} \\
\qquad\qquad (\tau_i'^{v_i'}, \phi_i) \;=\; \Gamma [\![x_i]\!] \, \forall \, i \in \{1, \ldots, n\} \\
\qquad\qquad \phi \;=\; \exists \, Y \, . \, (\phi_f \wedge (\wedge_{i=1}^{n} (v_i' = v_i))) \\
\qquad\qquad \pi_i \;=\; [\![\textit{assume} \; v_i = v_i';]\!] \\
\qquad\qquad \pi \;=\; \pi_1; \pi_2; \ldots; \pi_n; f(v_1, \ldots, v_n); \\
\qquad \textit{in} \;\; (\tau, \phi, \pi) \\
\mathcal{D} [\![\textbf{if} \; e_0 \; \textbf{then} \; e_1 \; \textbf{else} \; e_2]\!] \; \Gamma \, \psi \;=\; \\
\quad \textit{let} \; (\textbf{Bool}^{v}, \phi, \pi_0) \;=\; \mathcal{D} [\![e_0]\!] \; \Gamma \, \psi \\
\qquad (\tau_1{}^{v_1}, \phi_1, \pi_1) \;=\; \mathcal{D} [\![e_1]\!] \; \Gamma \; (\psi \wedge \phi \wedge (v = 1)) \\
\qquad (\tau_2{}^{v_2}, \phi_2, \pi_2) \;=\; \mathcal{D} [\![e_2]\!] \; \Gamma \; (\psi \wedge \phi \wedge (v = 0)) \\
\qquad \tau_3{}^{v_3} \;=\; \alpha \, (\tau_1{}^{v_1}) \\
\qquad Y \;=\; \{ v, v_1, v_2 \} \\
\qquad \phi_3 \;=\; \exists \, Y \, . \, \phi \wedge (((v_1 = v_3) \wedge (v = 1) \wedge \phi_1) \\
\qquad\qquad\qquad \vee ((v_2 = v_3) \wedge (v = 0) \wedge \phi_2)) \\
\qquad \pi \;=\; \pi_0; ((\textit{assume} \; v = 1; \; \pi_1; \; \textit{assume} \; v_3 = v_1;) \\
\qquad\qquad\qquad [\!] \, (\textit{assume} \; v = 0; \; \pi_2; \; \textit{assume} \; v_3 = v_2)) \\
\quad \textit{in} \;\; (\tau_3, \phi_3, \pi) \\
\mathcal{D} [\![\textbf{let} \; x = e_1 \; \textbf{in} \; e_2]\!] \; \Gamma \, \psi \;=\; \\
\qquad \textit{let} \; (\tau_1, \phi_1, \pi_1) \;=\; \mathcal{D} [\![e_1]\!] \; \Gamma \, \psi \\
\qquad\qquad (\tau, \phi_2, \pi_2) \;=\; \mathcal{D} [\![e_2]\!] \; \Gamma[(\tau_1, \phi_1)/x] \; \psi \\
\qquad\qquad Y \;=\; \textit{fv}(\tau_1) \\
\qquad\qquad \phi \;=\; \exists \, Y \, . \, \phi_1 \wedge \phi_2 \\
\qquad \textit{in} \;\; (\tau, \phi, \pi_1; \pi_2)
\end{array}
$$

Figure 13. Translation to PGCL with context information

variables expressed in ϕ. We define $wpc.\pi.\Phi$ to be the weakest pre-condition of π with respect to Φ, where π is obtained by executing $\mathcal{D}$ on e. That is to say,

$$wpc.\pi.\Phi = \forall Y.wp.\pi.\Phi$$

where Y is all free variables in the formula, excluding the input variables of π. In order to show that $wpc.\pi.\Phi = \forall X \, . \, Ctx(e) \Rightarrow \Phi$, we just need to show that $\forall Y.\, wp.\pi.\Phi = \forall \, X \, . \, Ctx(e) \Rightarrow \Phi$, for some X and Y which captures all the free variables in the respective formulae, except the input variables (since both formulae are expressed in terms of size variables, their input variables will be the same.)

The cases for variables and constants are trivial. We prove here the other cases:

case $[\![f \ (x_1, \ldots, x_n)]\!]$:

$$
\begin{aligned}
&\forall X.\, wp.(\pi_1; \ldots; \pi_n; f(v_1, \ldots, v_n)).\Phi \\
&= \forall X.\, wp.(\pi_1; \ldots; \pi_n).(wp.(f(v_1, \ldots, v_n)).\Phi \\
&= \forall X.\, wp.(\pi_1; \ldots; \pi_n).(Ctx(f(v_1, \ldots, v_n)) \Rightarrow \Phi) \\
&= \forall X.\, wp.(\pi_1; \ldots; \pi_n).(\phi_f \Rightarrow \Phi) \\
&= \forall X.\, (v_1 = v_1') \Rightarrow (\ldots (v_n = v_n') \Rightarrow (\phi_f \Rightarrow \Phi))) \\
&= \forall X.\, ((\wedge_{i=1}^{n} (v_i = v_i') \wedge \phi_f) \Rightarrow \Phi) \\
&= \forall X' \forall Y.\, ((\wedge_{i=1}^{n} (v_i = v_i') \wedge \phi_f) \Rightarrow \Phi) \\
&\qquad \text{where } Y = \cup_{i=1}^{n} v_i \text{ and } X = X' \cup Y \\
&= \forall X'.\, ((\exists Y.\, (\wedge_{i=1}^{n} (v_i = v_i') \wedge \phi_f)) \Rightarrow \Phi) \qquad \text{since } Y \cap fv(\Phi) = \emptyset \\
&= \forall X'.\, (Ctx([\![f \ (x_1, \ldots, x_n)]\!]) \Rightarrow \Phi)
\end{aligned}
$$

case $[\![\textbf{if} \ e_0 \ \textbf{then} \ e_1 \ \textbf{else} \ e_2]\!]$:

$$
\begin{aligned}
&wp.\pi_0; ((\mathit{assume}\ v = 1;\ \pi_1;\ \mathit{assume}\ v_3 = v_1;) \\
&\qquad [\!]\ (\mathit{assume}\ v = 0;\ \pi_2;\ \mathit{assume}\ v_3 = v_2)).\Phi \\
&= \forall X.\, wp.\pi_0.(((v = 1) \wedge \phi_1 \wedge (v_3 = v_1)) \Rightarrow \Phi \\
&\qquad\qquad \wedge\ wp.(\mathit{assume}\ v = 0;\ \pi_2;\ \mathit{assume}\ v_3 = v_2).\Phi) \\
&= \forall X.\, wp.\pi_0.(((v = 1) \wedge \phi_1 \wedge (v_3 = v_1)) \Rightarrow \Phi \\
&\qquad\qquad \wedge\ ((v = 0) \wedge \phi_2 \wedge (v_3 = v_2)) \Rightarrow \Phi) \\
&= \forall X.\, wp.\pi_0.(((v = 1) \wedge \phi_1 \wedge (v_3 = v_1) \\
&\qquad\qquad \vee (v = 0) \wedge \phi_2 \wedge (v_3 = v_2)) \Rightarrow \Phi) \\
&= \forall X.\, \phi \Rightarrow (((v = 1) \wedge \phi_1 \wedge (v_3 = v_1) \\
&\qquad\qquad \vee (v = 0) \wedge \phi_2 \wedge (v_3 = v_2)) \Rightarrow \Phi) \\
&= \forall X.\, (\phi \wedge ((v = 1) \wedge \phi_1 \wedge (v_3 = v_1) \\
&\qquad\qquad \vee (v = 0) \wedge \phi_2 \wedge (v_3 = v_2))) \Rightarrow \Phi) \\
&= \forall X' \forall Y.\, (\phi \wedge ((v = 1) \wedge \phi_1 \wedge (v_3 = v_1) \\
&\qquad\qquad\quad \vee (v = 0) \wedge \phi_2 \wedge (v_3 = v_2))) \Rightarrow \Phi) \\
&\qquad \text{where } Y = \{ v, v_1, v_2 \},\ v \in fv(\phi), \text{ and } X = X' \cup Y \\
&= \forall X'.\, (\exists Y.\, (\phi \wedge ((v = 1) \wedge \phi_1 \wedge (v_3 = v_1) \\
&\qquad\qquad\qquad \vee (v = 0) \wedge \phi_2 \wedge (v_3 = v_2)))) \Rightarrow \Phi) \\
&\qquad\quad \text{since } Y \cap fv(\Phi) = \emptyset \\
&= \forall X'.\, (Ctx([\![\textbf{if} \ e_0 \ \textbf{then} \ e_1 \ \textbf{else} \ e_2]\!]) \Rightarrow \Phi)
\end{aligned}
$$

case $[\![\textbf{let} \ x = e_1 \ \textbf{in} \ e_2]\!]$:

$$
\begin{aligned}
&\forall X.\, wp.\pi_1; \pi_2.\Phi \\
&= \forall X.\, wp.\pi_1.(wp.\pi_2.\Phi) \\
&= \forall X.\, (\phi_1 \Rightarrow (\phi_2 \Rightarrow \Phi)) \\
&= \forall X.\, ((\phi_1 \wedge \phi_2) \Rightarrow \Phi) \\
&= \forall X' \forall Y.\, ((\phi_1 \wedge \phi_2) \Rightarrow \Phi) \\
&\quad \text{where } Y = fv(\tau_1),\ \mathcal{D}[\![e_1]\!] \text{ evaluates to } (\tau_1, \phi_1, \pi_1), \text{ and } X = X' \cup Y \\
&= \forall X'.\, ((\exists Y.\, (\phi_1 \wedge \phi_2)) \Rightarrow \Phi) \qquad \text{since } Y \cap fv(\Phi) = \emptyset \\
&= \forall X'.\, (Ctx([\![\textbf{let} \ x = e_1 \ \textbf{in} \ e_2]\!]) \Rightarrow \Phi)
\end{aligned}
$$

Thus, we have $wpc.P'.\Phi = \forall X\ .\ Ctx(P) \Rightarrow \Phi$. □