A General Approach for Run-Time Specialization and its Application to C

Charles Consel François Noël
University of Rennes / Irisa
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{consel,fnoel}@irisa.fr

Abstract

Specializing programs with respect to run-time invariants is an optimization technique that has shown to improve the performance of programs substantially. It allows a program to adapt to execution contexts that are valid for a limited time.

Run-time specialization is being actively investigated in a variety of areas. For example, recently, major operating system research projects have been focusing on run-time specialization as a means to obtain efficiency from highly extensible and parameterized systems.

This paper describes a general approach to run-time specialization. For a given program and a declaration of its run-time invariants, it automatically produces source templates at compile time, and transforms them so that they can be processed by a standard compiler. At run time, only minor operations need to be performed: selecting and copying templates, filling holes with run-time values, and relocating jump targets. As a consequence, run-time specialization is performed very efficiently and thus does not require the specialized code to be executed many times before its cost is amortized.

Our approach improves on previous work in that: (1) templates are automatically produced from the source program and its invariants, (2) the approach is not machine dependent, (3) it is formally defined and proved correct, (4) it is efficient, as shown by our implementation for the C language.

1 Introduction

Specializing programs at run time with respect to dynamic invariants is an optimization technique that has already been explored in various areas such as operating systems [16] and graphics [14]. This technique is aimed at adapting programs to execution contexts by using run-time invariants.

In the context of file system operations, examples of run-time invariants include the type of the file being opened, the device where it resides, and whether it is exclusively read. When a file is being opened, at run time, invariants become available and can be exploited to specialize read and/or write routines. As reported by Pu *et al.* this specialization eliminates redundant interpretation of data structures and yields significant improvements [15].

In fact, various forms of run-time specializations have been studied on practical systems, and substantial improvements have been reported. Locanthi *et al.*, for example, applied specialization to the bitblit procedure [14, 12]; their specialized code ran about 4 times faster than a generic implementation. In the area of operating system, Massalin and Pu designed an operating system which utilized run-time specialization as a fundamental technique to optimize a wide variety of system components. They report speedup factors that range from 2 to 40 depending on the system component considered [13].

Although various forms of run-time specializations have undoubtedly been shown to improve substantially the performance of programs, the specialization process has always been done manually [9]. The usual approach consists of defining *code templates*, that is, code fragments parameterized with respect to run-time values. Then at run time, templates are linked together depending on the control flow, and holes (*i.e.,* template parameters) are filled with run-time values [10]. To minimize the cost of run-time specialization, templates are often represented in a binary form to avoid invoking an assembler, or even more expensive, a complete compiler, at run time.

While the idea of run-time specialization is certainly attractive, considering the degree of improvement it can yield, the approaches explored so far have fundamental drawbacks.

- They are manual. Usually templates are written by the programmer either directly in some low level language, or using some syntactic facilities [6].
- They are not clearly defined. Although existing approaches have shown their effectiveness, the process of run-time specialization has always been presented as a black-box; only the functionalities were described not the techniques.
- They are not portable. When templates are written in assembly language, they are limited to a given processor. Often, templates have to be optimized manually to obtain good performance.
- They are error-prone. Because templates are directly written by the programmer in a low-level language, errors may easily be introduced.

In this paper we present a general approach for run-time specialization and its application to the C programming language. Our approach can be decomposed in the following main stages. At compile time, a program is analyzed for a given context of invariants declared by the programmer. This analysis determines the program transformation to be performed for every syntactic construct in the program. This information is used by a subsequent analysis to produce a safe approximation of the possible specializations of this program, in the form of a tree grammar. Then, this tree grammar is used to generate templates *automatically* at the source level. These templates capture the dynamic computations, that is, the computations that rely on data that vary. Once compiled by a standard compiler (in the case of the C language), various linking information (*i.e.,* labels and holes) is collected from the compiled templates. Finally, the parts of the program corresponding to the static computations, *i.e.,* the computations that rely on the invariants, are compiled; they represent the *run-time specializer*. When it is executed at run time, in addition to computing invariants, the run-time specializer also selects templates depending on the control flow, relocates jump targets, and fills template holes with the invariant values.

Our approach has many advantages compared to the existing ones.

- It is automatic. Templates are automatically generated from a description of the possible specializations of a program, itself produced by an analysis.
- It is formally based. We have formally defined the approach for a subset of an imperative language and proved it correct.
- It is general. In principle, our approach applies to a variety of languages, from imperative to applicative ones.
- It is portable. In our approach, most of the specialization process is in fact performed at the source level. Only minor operations in the linking phase of templates need to be ported. These operations are limited to collecting locations of template holes and jump labels within templates.
- It is efficient. We have applied the approach to the C language: an implementation of a run-time specializer of C programs has been developed. It has been used on various kinds of programs. The run-time specialization process incurs a negligible overhead. Preliminary experimentation shows that on procedures exhibiting a clear interpretive layer (*e.g.,* variations of printf), run-time specialized code requires as little as 3 runs to amortize the cost of specialization, and it executes 5 times faster than the non-specialized version.

Our run-time specialization approach is based, in part, on partial evaluation technology [8, 3]. In fact, it is integrated in a complete partial evaluation system for C programs that performs compile-time specialization as well as run-time specialization [4]; this aspect is further discussed in Section 2. This system has been applied to various kinds of programs such as operating system code.

Plan. In Section 2, the underlying concepts of partial evaluation are reviewed. In Section 3, the approach and its main components are described; examples are used to illustrate the presentation. In Section 4, the approach is formally defined and proved correct. Section 5 then discusses the related work. Finally, Section 6 gives some concluding remarks, and outlines the future directions of this work.

2 Partial Evaluation

Partial evaluation is a program transformation technique aimed at specializing a program with respect to some parts of its input [8, 3]. There are two main strategies to perform partial evaluation. The first strategy, called *on-line*, consists of specializing a program in a single pass. As the program gets processed, the program transformations are determined and performed. Because program transformation occurs in the presence of concrete values, on-line partial evaluation achieves a high-degree of specialization.

The second strategy to partial evaluation, called *off-line*, is composed of two parts: preprocessing and specialization. For a given program and a description of its input (known or unknown), the preprocessing phase essentially compiles the specialization phase. It does so by determining a program transformation for each syntactic construct in the program. Then, the specialization phase is performed with respect to some partial input value. This process is solely guided by the information produced by the preprocessing phase. As a consequence, specialization is efficient.

Whether on-line or off-line, partial evaluation has always been studied and understood as a source-to-source program transformation. In this paper, we introduce an approach that goes beyond this view. We propose to use partial evaluation as a basis for run-time specialization. In fact, this work is part of a complete partial evaluation system which specializes C programs at compile time as well as at run time [4]. Let us briefly outline the salient features of this system.

Our partial evaluation system is based on an off-line strategy. The preprocessing phase mainly consists of an alias analysis, a binding-time analysis, and an action analysis. The alias analysis is needed because of the pointer facilities offered by the C language. The binding-time analysis determines the binding-time property of the variables in a program, given a program and a description of its context (*i.e.,* global variables and formals are declared as *static*/known or *dynamic*/unknown). While the binding-time analysis determines what to do for each syntactic construct in a program, the action analysis determines how to do it [2]. In other words, binding-time information is used to determine what *specialization action* (*i.e.,* program transformation) should be performed. A small subset of the actions used for the C language is presented in Section 4.

Once the actions of a program are produced, various back-ends can exploit this information. Firstly, they can be interpreted; this situation corresponds to a specializer. Secondly, they can be compiled to produce a dedicated specializer (also called a generating extension [8]). Finally, actions can be used to achieve run-time specialization. The last alternative comes from the fact that actions define program transformations and thus can be used as a basis to determine what specialized programs an action-analyzed program can yield. In fact, in our approach, an analysis of an action-analyzed program is performed to determine an approximation of this set of possible specialized programs; this set is described as a tree grammar.

3 An Approach to Run-Time Specialization

As mentioned in Section 2, run-time specialization is one

int f(**int** x, **int** y) {	**int** f(**int** x, **int** y) {
int l;	**int** l;
$l = 2 * x$;	$(l = 2 * x)^{ev}\ ;^{red}$
if ($l == 2$)	$\mathbf{if}^{red}$ ($l == 2$ $)^{ev}$
$l = l + y$;	$l^{id} =^{reb} l^{ev} +^{reb} y^{id}\ ;^{reb}$
else $l = y * x$;	**else** $l^{id} =^{reb} y^{id} *^{reb} x^{ev}\ ;^{reb}$
return l;	(**return** $l)^{id}$;
}	}
(a) Source program	(b) Action-analyzed program (x = static, y = dynamic)

Figure 1: An example program

of the three ways of exploiting action-analyzed programs. This section presents an approach to run-time specialization based on actions.

Let us explain in detail how a program gets annotated with actions by taking a concrete example. Actions are indeed a key aspect of our approach: it is the starting point of the run-time specialization process.

Figure 1-a presents a source program, written in the C language. Figure 1-b shows this program annotated with actions. For readability, the action-analyzed program is represented in concrete syntax and decorated with actions. Let us describe how actions are determined for procedure f in the example program, assuming that its first parameter is static and its second parameter dynamic.

The first command in the procedure is assigned action *eval* (ev). This action annotates completely static program fragments. Such program fragments represent computations that solely depend on available data. Assuming that the symbol ';', at the end of the first command, is a sequence construct, then it is assigned the action *reduce* (red) because the first command will be evaluated away and thus the sequence command will be reduced. Likewise, the conditional command can also be reduced because the value of the test expression can be determined at specialization-time. Still, the branches of the conditional command have to be rebuilt. More specifically, the assignment in the true branch has to be *rebuilt* (reb) because the right-hand side does not partially evaluate to a constant. This is caused by variable y which is dynamic. As such, this variable represents a completely dynamic code fragment; it is annotated with action *identity* (id). This action denotes code fragments that can be reproduced verbatim in the specialized program. A similar situation occurs in the false branch of the conditional command. Finally, the return command is globally annotated with id since it is uniformly dynamic.

To specialize a program at run time based on its actions, a naïve approach would simply consist of postponing specialization until run time, that is, when the specialization values become available. Then, the specialized code would be compiled and dynamically linked to the running executable. The obvious drawback of this approach is the cost of compilation which would require the run-time specialized program to be run many times to amortize the cost of specialization, compilation and linking.

In fact, the reason why the compilation of a particular specialized program has to be postponed until run time is because we do not know the set of possible specializations an action-analyzed program can yield. If we knew such a set, or a description of it, then it could be processed at compile time instead of run time. Unfortunately, the set of all possible specializations is in general infinite (because of loop unrolling, for example).

3.1 Using Tree Grammars

A traditional way to finitely represent an infinite set of trees is to use tree grammars. However, determining the exact set of the possible specializations of an action-analyzed program is undecidable in general, since specialization values are unknown at compile time. Yet, an approximation can be defined; it corresponds to the least superset of the exact set. It is safe to consider a tree grammar that describes more specializations than the actual ones if they are ignored during run-time specialization; more precisely, if no execution context leads to these specializations.

We have developed an analysis aimed at computing a tree grammar, called *specialization grammar* in this context, which represents a safe approximation of the set of all possible specializations of an action-analyzed program. Let us consider an example of a specialization grammar. Figure 2-a redisplays the action-analyzed procedure f and Figure 2-b shows its corresponding specialization grammar. Like action-analyzed programs, specialization grammars are represented in concrete syntax.

The first rule F describes the possible specializations of procedure f. Unlike compile-time specialization, when a procedure is specialized at run time, it does not need to be renamed. Indeed, during execution, templates have a binary format. Only code addresses are manipulated. Since local variable l is involved in some dynamic computations, it is residual, and thus its declaration remains in the specialized program. Directly following this declaration, instead of the first command of the original procedure, the non-terminal S occurs. This non-terminal defines the specializations of the conditional command. In fact, the first command in f is not part of the specialization grammar because it is completely static (ev); consequently, it will be evaluated at specialization-time. Next to the occurrence of the non-terminal S, the return command appears. It is identical to the command in the original program because it is completely dynamic (id). As for the conditional command described by rule S, it will be reduced at run time since its test expression is purely static. As a result, rule S is composed of two alternatives, one for each branch. Each branch is an assignment to be rebuilt at run time. However, each right-hand side of these assignments includes a completely static expression: variables l and x. The integer values resulting from their run-time evaluation are described by the

$$
\begin{array}{l}
\textbf{int } f(\textbf{int } x, \textbf{int } y) \ \{ \\
\quad \textbf{int } l; \\
\quad (l = 2 * x)^{ev} \ \ ;^{red} \\
\quad \textbf{if}^{red} \ (\ l == 2 \ )^{ev} \\
\quad\quad l^{id} =^{reb} l^{ev} +^{reb} y^{id} \ \ ;^{reb} \\
\quad \textbf{else } l^{id} =^{reb} y^{id} *^{reb} x^{ev} \ \ ;^{reb} \\
\quad (\textbf{return } l)^{id} \ ; \\
\}
\end{array}
$$

(a) Action-analyzed program

$$
\begin{array}{lcl}
F & \rightarrow & \textbf{int } f\_t(\textbf{int } y) \ \{ \\
 & & \quad \textbf{int } l; \\
 & & \quad S \\
 & & \quad \textbf{return } l; \\
 & & \} \\
 & & \\
S & \rightarrow & l = Int + y; \\
 & | & l = y * Int;
\end{array}
$$

(b) Specialization grammar

Figure 2: Specialization grammar generated from an action-analyzed procedure

generic terminal *Int*; it is a placeholder for integer values.

At this stage it is important to notice that a special-purpose compiler could be developed to process right-hand sides of specialization grammar rules. In other words, such a tool could compile incomplete syntax trees parameterized with constant values. Compilation would be done statically and thus run-time specialization would mainly amount to assembling binary fragments and instantiating them with respect to run-time values. Although this method is possible, it requires one to develop a complete compiler. This compiler would necessitate time and effort to be competitive with advanced optimizing compilers currently available. A better approach would consist of modifying an existing compiler. However, real-size compilers are not as modular as they claim to be: significant modifications may propagate throughout most of the compilation system. Such modifications may therefore not be necessarily much simpler than the previous approach. An even better approach consists of using an existing compiler as is. This is discussed in the next sections.

3.2 Introducing Templates

An existing compiler can be used to process right-hand sides of specialization grammars. In this section we present a transformation process aimed at converting these right-hand sides into source code fragments parameterized with run-time values. We call these fragments *source templates*. Template parameters are often called *holes* in the literature [10]. At run time, part of the specialization process consists of physically replacing these parameters by values. In other words, template holes are filled with run-time values. The resulting object is called an *instance* of the template.

Transforming specialization grammars into source level templates mainly amounts to unparsing the right-hand side of the grammar rules and delimiting individual templates. The former task is fairly straightforward. The only interesting aspect is concerned with the treatment of generic terminals. This representation for run-time values is transformed into holes. The concrete representation of a hole depends on both the language and the compiler being used. To abstract over these issues, holes are just given a unique name within brackets (for example $[h_1]$ in Figure 3).

Delimiting templates can be done in several ways. Let us present two approaches.

The first approach consists of creating one template per right-hand side in a specialization grammar. For example, based on the specialization grammar presented in Figure 2-b, the first approach would yield three templates as shown in Figure 3-a: one for procedure f, and one per alternative in the conditional command. Just like the right-hand side of F in the specialization grammar includes non-terminal S, its corresponding template includes a reference to other templates. The template to be selected cannot be determined at compile time since it depends on the value of the test expression of the conditional command. Therefore, a placeholder (that is, some reserved space) is introduced to insert the selected template.

Although conceptually simple, this approach may be costly in practice. Indeed, it assumes that the physical layout of template t_1 at run time includes enough space to insert either template t_2 or template t_3. If they have different sizes, the size of largest template is used. As a result, placeholders for templates may have a large size. A more important drawback occurs if loops are unrolled. In this case, the size of the unrolled loop cannot be determined at compile time.

The second approach is aimed at eliminating nested templates such that no space be reserved to insert templates. To do so, when the right-hand side of a specialization grammar rule includes a non-terminal, a template is created before and after this non-terminal. This approach is illustrated by Figure 3-b. Template t_1 represents a first fragment of the specialized version of procedure f. Then, either template t_2 or t_3 is appended. Finally, template t_4 completes a specialized procedure.

Notice that for the formal definition of the run-time specialization process, the first approach is used to simplify the presentation and abstract over these implementation issues.

3.3 Compiling Templates Statically

Once templates are identified and transformed into concrete syntax, they can be compiled. Because they are available at compile time, they can be compiled then. They become *object templates*. Of course, the way templates are compiled depends on the language in which they are written, and the compiler which is used.

In this section we discuss the general issues arising for template compilation, mostly independently of a specific language or compiler.

So far, the templates of a procedure have been described as separate entities. However, if templates were to be compiled separately, the quality of the code would be poor since the compilation process would not take advantage of the context in which they appear. Some compilation aspects such as register allocation and instruction scheduling would undoubtedly suffer from this situation. To circumvent this problem, our approach consists of constructing a source code

```
int f_t(int y) {        t1
    int l;
    t2 or t3
    return l;
}

l = [h1]+y;             t2

l = y * [h2];           t3
```

(a) Approach 1

```
int f_t(int y) {        t1
    int l;

l = [h1]+y;             t2

l = y * [h2];           t3

    return l;           t4
}
```

(b) Approach 2

Figure 3: Templates for procedure f

```
int f_t(int y) {               t1
    int l;
    switch unknown {
      case 1:
            l = [h1]+y;        t2
            break;
      case 2:
            l = y * [h2];      t3
            break;
    }
    return l;                  t4
}
```

Figure 4: Source representation of templates

```
void rt_spec_f(int x) {
    int l;

    dump_template(t1);
    l = 2 * x;
    if ( l == 2 ) {
        dump_template(t2);
        instantiate_hole(t2, l);
    } else {
        dump_template(t3);
        instantiate_hole(t3, x);
    }
    dump_template(t4);
}
```

Figure 5: Run-time specializer for f

that combines all the templates and still expresses the unknowns as far as how exactly these templates can be assembled at run time. A concrete example of this transformation is presented in Figure 4.

As can be noticed the source representation of templates for procedure f follows the structure of the specialization grammar. In particular, because we do not know prior to run time which alternative of the conditional command will be included in the specialized version of procedure f, both alternatives are included in a switch command whose test value is unknown (variable *unknown*) to the compiler. This layout is directly derived from the specialization grammar. Even though there is this unknown, the compiler can still process the templates globally, in that it knows the possible combinations that can occur. In fact, the source representation of templates includes some form of markers around templates so that they can be identified and extracted from the object code. Object templates are used at run time by the specializer.

Finding an appropriate representation for template holes is an issue that depends on both the language and the compiler being studied.

Once templates are compiled, information from the resulting object code must be collected: the address of template holes needs to be recorded so that the run-time specializer knows where values need to be installed. Also, template addresses have to be determined so that jumps can be relocated if needed.

3.4 Producing the Run-Time Specializer

Now that templates have been generated, compiled and extracted from the object code, and that information needed to instantiate them has been collected, we are ready to produce the *run-time specializer*. This procedure consists of eval fragments interleaved with operations aimed at selecting and dumping templates, filling holes with run-time values, and relocating jump targets. The run-time specializer is generated based on an action-analyzed program.

Figure 5 displays the run-time specializer for procedure f. The control flow of this procedure can be seen as a subset of the control flow of the original procedure in the sense that only the static parts of the original control flow graph appear in the run-time specializer.

Since parameter x in the original procedure was declared as static, it appears as a parameter of the run-time specializer. Local variable l was involved both in static and dynamic computations. Therefore it appears in both a template and the run-time specializer. The first operation of the run-time specializer is to dump template t_1, which is the header of the specialized procedure. The first command of the original procedure can then be executed since it is purely static. Next, the conditional command is executed. The test expression can be fully evaluated; the resulting value determines whether the first or the second template should be dumped. The dumped template is then instantiated with the appropriate run-time value. Finally template t_4 is dumped; it corresponds to the purely dynamic return command, and thus does not require any instantiation.

As can be noticed, the operations to perform the actual specialization are very simple and introduce little overhead at run time. Relocation of jump targets and hole filling are compiled. Copying of templates can be implemented very efficiently on some processors provided their memory layout is carefully done.

The result of an invocation of the run-time specializer is a specialized code ready to be used. In our implementation, the last operation of the run-time specializer consists of returning the address of the specialized code. For a procedure, it returns a procedure pointer which can then be invoked.

4 Semantic Definition of Run-Time Specialization

In this section, an imperative languague is introduced and its semantics is defined. Then, a set of specialization actions for this language, as well as their semantics, are presented. Also, the semantic definition of the process of generating run-time specializers is given. Finally, the correctness criterion for this latter process is stated. It establishes that specializing programs by interpreting actions, or by evaluating the run-time specializer yields the same specialized program, given the same specialization values.

Even though this presentation covers a simple imperative language and a small set of actions, it still addresss the important steps of the run-time specialization process. Because this presentation is done in a denotational framework, it abstracts over implementation details and focuses on conceptual aspects.

4.1 The Language

Variations of the language being studied (and their semantic definition) are used in this presentation. To distinguish each of them, syntactic domains and variables ranging over these domains are indexed by the abbreviated name of the variation (*e.g.,* $c^i \in Com^i$), and similarly for valuation functions.

The syntax of the imperative language being studied is displayed in Figure 6. The first part of the figure (Com^i and Exp^i) defines the language to be handled by the specializer. This initial language consists of commands (empty commands noted **Nop**, assignments, sequences, and conditionals) and expressions (variables, constants, and primitive calls).

To reason about run-time specialization, the initial language is extended. To motivate these extensions, let us discuss some issues involved in modeling run-time specialization in a denotational framework.

First, as can be expected the denotational definition of run-time specialization does not manipulate object templates. Instead, it manipulates source templates. More precisely, since source templates are essentially in a one-to-one correspondence with the right-hand sides of grammar rules, the latter ones will now be manipulated by the run-time specialization process.

As a consequence of this change, instead of dumping templates for each non-terminal and instantiating templates with constant values, run-time specialization now substitutes non-terminals by their right-hand side, and generic terminals (encoded as holes) by constant values. Two extensions to our initial language make it possible to perform these operations. Construct $\mathbf{Rule}(s, c^{rhs})$ allows a non-terminal s to be replaced by its right-hand side c^{rhs}. Construct $\mathbf{Inst}(h, e)$ substitutes a hole h by a constant resulting from the evaluation of an expression e. The extended language is defined by domains Com and Exp.

$$
\begin{array}{lll}
x \in Id & & \text{Identifiers} \\
n \in Num & & \text{Numbers} \\
o \in Oper & & \text{Binary operators} \\
h \in Holes = \{h_1, \ldots, h_m\} & & \text{Holes} \\
s \in Nterms = \{s_1, \ldots, s_n\} & & \text{Non-terminals} \\
c^i \in Com^i & ::= & \mathbf{Nop} \\
 & \mid & \mathbf{Assign}(x, e^i) \\
 & \mid & \mathbf{Seq}(c_1^i, c_2^i) \\
 & \mid & \mathbf{Cond}(e^i, c_1^i, c_2^i) \\
e^i \in Exp^i & ::= & \mathbf{Var}(x) \\
 & \mid & \mathbf{Cst}(n) \\
 & \mid & \mathbf{Call}(o, e_1^i, e_2^i) \\
c \in Com & ::= & \mathbf{Nop} \\
 & \mid & \mathbf{Assign}(x, e) \\
 & \mid & \mathbf{Seq}(c_1, c_2) \\
 & \mid & \mathbf{Cond}(e, c_1, c_2) \\
 & \mid & \mathbf{Rule}(s, c^{rhs}) \\
 & \mid & \mathbf{Inst}(h, e) \\
e \in Exp = Exp^i & & \\
c^{rhs} \in Com^{rhs} & ::= & \mathbf{Nop} \\
 & \mid & \mathbf{Assign}(x, e^{rhs}) \\
 & \mid & \mathbf{Seq}(c_1^{rhs}, c_2^{rhs}) \\
 & \mid & \mathbf{Cond}(e^{rhs}, c_1^{rhs}, c_2^{rhs}) \\
 & \mid & \mathbf{Nterm}(s) \\
e^{rhs} \in Exp^{rhs} & ::= & \mathbf{Var}(x) \\
 & \mid & \mathbf{Cst}(n) \\
 & \mid & \mathbf{Call}(o, e_1^{rhs}, e_2^{rhs}) \\
 & \mid & \mathbf{Hole}(h)
\end{array}
$$

Figure 6: Language syntax

Right-hand sides of grammar rules are defined by domains Com^{rhs} and Exp^{rhs}. Just as templates can be nested, right-hand side terms (rhs-terms) may include non-terminals ($\mathbf{Nterm}(s)$). Also, expressions may include holes ($\mathbf{Hole}(h)$).

The end result of run-time specialization now corresponds to the abstract syntax of the specialized program, without non-terminals nor holes.

The model we just described does not contradict the fact that source templates are available at compile time and can thus be compiled prior to run-time to achieve efficient specialization in practice.

4.2 Semantic Definition of the Extended Language

In this section the denotational semantics of the extended language is defined. It is not necessary to define the denotational semantics of the initial language since it is a subset of the extended one.

The semantic domains as well as the valuation functions are displayed in Figure 7.

Notice that the process of substituting non-terminals by their right-hand side, and holes by values is noted '$\longleftarrow$'.

As discussed in the previous section, we define the semantics of run-time specialization at the abstract syntax level. To do so, we have introduced extra constructs (**Rule** and **Inst**) to build a specialized program by repeated substitutions. But we also need to define a place where the program being specialized can be stored and incrementally built. To this end, a special identifier '§' is introduced; the store maps

$$
\begin{array}{llcl}
i \in Int & & & \text{Integer values} \\
f \in Fun_2 = Int \times Int \rightarrow Int & & & \text{Binary, integer functions} \\
\sigma, \delta \in Store = Id \rightarrow (Int + Com^{rhs})_\perp & & & \\
\mathcal{C} : Com \rightarrow Store \rightarrow Store & & & \\
\mathcal{C}[\![\mathbf{Nop}]\!] & \sigma & = & \sigma \\
\mathcal{C}[\![\mathbf{Assign}(x, e)]\!] & \sigma & = & \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma] \\
\mathcal{C}[\![\mathbf{Seq}(c_1, c_2)]\!] & \sigma & = & \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma) \\
\mathcal{C}[\![\mathbf{Cond}(e, c_1, c_2)]\!] & \sigma & = & \text{if } \mathcal{E}[\![e]\!]\sigma \text{ then } \mathcal{C}[\![c_1]\!]\sigma \text{ else } \mathcal{C}[\![c_2]\!]\sigma \\
\mathcal{C}[\![\mathbf{Rule}(s, c^{rhs})]\!] & \sigma & = & \sigma[\S \mapsto \sigma(\S)[s \leftarrow c^{rhs}]] \\
\mathcal{C}[\![\mathbf{Inst}(h, e)]\!] & \sigma & = & \sigma[\S \mapsto \sigma(\S)[h \leftarrow \mathcal{E}[\![e]\!]\sigma]] \\
& & & \\
\mathcal{E} : Exp \rightarrow Store \rightarrow Int & & & \\
\mathcal{E}[\![\mathbf{Var}(x)]\!] & \sigma & = & \sigma(x) \\
\mathcal{E}[\![\mathbf{Cst}(n)]\!] & \sigma & = & \mathcal{N}[\![n]\!] \\
\mathcal{E}[\![\mathbf{Call}(o, e_1, e_2)]\!] & \sigma & = & \mathcal{O}[\![o]\!](\mathcal{E}[\![e_1]\!]\sigma, \mathcal{E}[\![e_2]\!]\sigma) \\
& & & \\
\mathcal{N} : Num \rightarrow Int & & & \\
\mathcal{O} : Oper \rightarrow Fun_2 & & &
\end{array}
$$

Figure 7: Extended semantics

```
{
    rule(s0, ⟨|s1; return l;|⟩)
    l = 2 * x;
    if ( l == 2 ) {
        rule(s1, ⟨|l = [h1] + y;|⟩);
        inst([h1], l);
    }
    else {
        rule(s1, ⟨|l = y * [h2];|⟩);
        inst([h2], x);
    }
}
```

Figure 8: A run-time specializer written in the extended language

it to the specialized program being built. For a specialization grammar of a given program, the initial state of the specialization process consists of a store mapping identifier § to the right-hand side of the start symbol of the grammar.

Notice that holes and non-terminals are unique, as specified by the generator of run-time specializers (see Section 4.4).

Let us revisit the example of procedure f and examine the run-time specializer for its body; it is displayed in Figure 8. The declaration is omitted, and the return command is left for the sake of presentation although procedures are not included in the initial language.

Since identifier § is initially mapped to the right-hand side of the start symbol of the specialization grammar, as the run-time specializer executes, the non-terminals get replaced by their right-hand side, and holes get substituted by constants.

4.3 Semantic Definition of Specialization Actions

Now that the extended language is introduced, let us define the syntax and semantics of specialization actions. They represent the starting point of the run-time specialization process. The set of actions considered for this presentation

$$
\begin{array}{lcl}
c^a \in Com^a & ::= & \mathbf{Eval}(c^i) \\
& | & \mathbf{Id}(c^i) \\
& | & \mathbf{Rebassign}(x, e^a) \\
& | & \mathbf{Rebseq}(c_1^a, c_2^a) \\
& | & \mathbf{Redseq}(c_1^i, c_2^a) \\
& | & \mathbf{Rebcond}(e^a, c_1^a, c_2^a) \\
& | & \mathbf{Redcond}(e^i, c_1^a, c_2^a) \\
e^a \in Exp^a & ::= & \mathbf{Eval}(e^i) \\
& | & \mathbf{Id}(e^i) \\
& | & \mathbf{Rebcall}(o, e_1^a, e_2^a)
\end{array}
$$

Figure 9: Actions syntax

is displayed in Figure 9.

The meaning of all these actions has been discussed earlier except for **Rebseq**. This action is assigned to a sequence command to be rebuilt. Notice that eval and identity commands (and expressions) only involve elements of the initial language. Indeed, in either case these commands (and expressions) do not involve any specialization aspects and should thus be standard. A similar situation occurs for the first argument of both **Redseq** and **Redcond** which is purely static. The semantic definition of the actions is given in Figure 10.

As discussed above, the semantic of action **Redseq** requires the first command to be purely static; another action could be introduced to address the case when the second command is purely static.

Lastly, it is important to notice that the actions of a given program are assumed to be correct. Proving the correctness of actions is outside the scope of this paper. This issue is addressed by Consel and Khoo in the context of a functional language [5].

4.4 Generating Run-time Specializers

Given that the semantics of actions are defined, the remaining step is aimed at generating the run-time specializer from an action-analyzed program. This generator of run-

$$
\begin{array}{lcll}
\mathcal{C}^a : Com^a \rightarrow Store \rightarrow (Com^i \times Store) & & & \\
\mathcal{C}^a[\![\mathbf{Eval}(c^i)]\!] & \sigma & = & ([\![\mathbf{Nop}]\!], \mathcal{C}[\![c^i]\!]\sigma) \\
\mathcal{C}^a[\![\mathbf{Id}(c^i)]\!] & \sigma & = & ([\![c^i]\!], \sigma) \\
\mathcal{C}^a[\![\mathbf{Rebassign}(x, e^a)]\!] & \sigma & = & ([\![\mathbf{Assign}(x, \mathcal{E}^a[\![e^a]\!]\sigma)]\!], \sigma) \\
\mathcal{C}^a[\![\mathbf{Rebseq}(c_1^a, c_2^a)]\!] & \sigma & = & ([\![\mathbf{Seq}(c_1^i, c_2^i)]\!], \sigma'') \\
 & & & \quad \text{where } \; (c_1^i, \sigma') = \mathcal{C}^a[\![c_1^a]\!]\sigma \\
 & & & \qquad\qquad (c_2^i, \sigma'') = \mathcal{C}^a[\![c_2^a]\!]\sigma' \\
\mathcal{C}^a[\![\mathbf{Redseq}(c_1^i, c_2^a)]\!] & \sigma & = & \mathcal{C}^a[\![c_2^a]\!](\mathcal{C}[\![c_1^i]\!]\sigma) \\
\mathcal{C}^a[\![\mathbf{Rebcond}(e^a, c_1^a, c_2^a)]\!] & \sigma & = & ([\![\mathbf{Cond}(\mathcal{E}^a[\![e^a]\!]\sigma, c_1^i, c_2^i)]\!], \sigma'') \\
 & & & \quad \text{where } \; (c_1^i, \sigma') = \mathcal{C}^a[\![c_1^a]\!]\sigma \\
 & & & \qquad\qquad (c_2^i, \sigma'') = \mathcal{C}^a[\![c_2^a]\!]\sigma' \\
\mathcal{C}^a[\![\mathbf{Redcond}(e^i, c_1^a, c_2^a)]\!] & \sigma & = & \text{if } \mathcal{E}[\![e^i]\!]\sigma \text{ then } \mathcal{C}^a[\![c_1^a]\!]\sigma \text{ else } \mathcal{C}^a[\![c_2^a]\!]\sigma \\
 & & & \\
\mathcal{E}^a : Exp^a \rightarrow Store \rightarrow Exp^i & & & \\
\mathcal{E}^a[\![\mathbf{Eval}(e^i)]\!] & \sigma & = & [\![\mathbf{Cst}(\mathcal{E}[\![e^i]\!]\sigma)]\!] \\
\mathcal{E}^a[\![\mathbf{Id}(e^i)]\!] & \sigma & = & [\![e^i]\!] \\
\mathcal{E}^a[\![\mathbf{Rebcall}(o, e_1^a, e_2^a)]\!] & \sigma & = & [\![\mathbf{Call}(o, \mathcal{E}^a[\![e_1^a]\!]\sigma, \mathcal{E}^a[\![e_2^a]\!]\sigma)]\!]
\end{array}
$$

Figure 10: Semantic definition of actions

time specializers is defined as a non-standard interpretation of actions. For a given action-analyzed program, it produces two results: an rhs-term which corresponds to the unsubstituted specialized program, and a run-time specializer which includes substitution operations and eval fragments. The generator is defined in Figure 11.

Let us describe in detail the treatment of each action, starting with the commands. An eval command produces an rhs-term which consists of the empty command since a command which can be completely evaluated will not appear in the specialized program. As for the run-time specializer, it corresponds to the command itself since it can be completely evaluated. The inverse situation happens for identity commands.

Rebuilding an assignment means that this construct will be in the specialized program and thus is included in the resulting rhs-term. This rhs-term corresponds to the original assignment where eval expressions (in the right-hand side) have been replaced by holes. As for the run-time specializer, it is composed of the instantiation operations that may be needed to fill the holes in the right-hand side expression of the assignment.

Rebuilding a sequence command means that this construct will appear in the specialized program, and indeed, it is part of the resulting rhs-term. As for the run-time specializer, it is composed of the eval commands contained in the arguments of sequence command.

When reducing a sequence command, the generated rhs-term only contains the commands from the second argument of sequence to be rebuilt (the first argument can be completely evaluated). The run-time specializer is a sequence command which consists of the first argument of the original sequence, and the eval commands from the second argument of sequence.

Rebuilding a conditional command is very similar to rebuilding a sequence command; its description is thus omitted. The reduction of a conditional command involves a new aspect: it produces a fresh non-terminal as the rhs-term. This is due to the fact that, although the conditional command is known to be reduced, the branch to consider is unknown. Therefore, a non-terminal is introduced as a placeholder for the rhs-term of either branch. Consequently, the run-time specializer produced in this situation consists of a conditional to be evaluated whose branches substitute the fresh non-terminal by the rhs-term of the appropriate branch, in addition to executing the eval commands contained in the corresponding branch.

In the case of an eval expression, the result of its evaluation will be substituted for a hole at run time. Therefore, the analysis of such an expression produces a hole freshly generated as the rhs-term. As for the run-time specializer, it consists of an instantiation command aimed at replacing the hole by a value computed at specialization time.

When an identity expression is analyzed, it is reproduced verbatim as the rhs-term. As for the run-time specializer, it consists of the empty command since the expression is not processed during specialization.

Rebuilding a primitive call means that the rhs-term consists of this construct, the operator, and the rhs-term of each operand. The run-time specializer is a sequence construct composed of the instantiation commands caused by the possible eval expressions included in the call arguments.

4.5 Correctness

Proving correct the process of generating run-time specializers consists of showing that, for an action-analyzed program and some specialization values, the specialized program produced by interpreting actions is the same as the one produced by executing the run-time specializer using the same specialization values.

This statement is formally expressed in the following theorem.

Theorem 1 $(\forall c^a \in Com^a)(\forall \sigma \in Store)$
Let $(c^{rhs}, c) = \mathcal{C}^a_{gen}[\![c^a]\!]$
Then, $\sigma' = \mathcal{C}[\![c]\!]\sigma[\S \mapsto c^{rhs}] \Rightarrow (\sigma'(\S), \bar{\sigma'}) = \mathcal{C}^a[\![c^a]\!]\bar{\sigma}$

Where $\forall \sigma \in Store, \bar{\sigma} = \sigma[\S \mapsto \perp]$
The proof is included in Appendix A.

$$
\begin{array}{lcl}
\mathcal{C}^a_{gen} : Com^a \rightarrow (Com^{rhs} \times Com) & & \\
\mathcal{C}^a_{gen}[\![\mathbf{Eval}(c^i)]\!] & = & ([\![\mathbf{Nop}]\!], [\![c^i]\!]) \\
\mathcal{C}^a_{gen}[\![\mathbf{Id}(c^i)]\!] & = & ([\![c^i]\!], [\![\mathbf{Nop}]\!]) \\
\mathcal{C}^a_{gen}[\![\mathbf{Rebassign}(x, e^a)]\!] & = & ([\![\mathbf{Assign}(x, e^{rhs})]\!], [\![c]\!]) \\
 & & \quad \text{where} \quad (e^{rhs}, c) \; = \; \mathcal{E}^a_{gen}[\![e^a]\!] \\
\mathcal{C}^a_{gen}[\![\mathbf{Rebseq}(c^a_1, c^a_2)]\!] & = & ([\![\mathbf{Seq}(c^{rhs}_1, c^{rhs}_2)]\!], [\![\mathbf{Seq}(c_1, c_2)]\!]) \\
 & & \quad \text{where} \quad (c^{rhs}_1, c_1) \; = \; \mathcal{C}^a_{gen}[\![c^a_1]\!] \\
 & & \quad \phantom{\text{where}} \quad (c^{rhs}_2, c_2) \; = \; \mathcal{C}^a_{gen}[\![c^a_2]\!] \\
\mathcal{C}^a_{gen}[\![\mathbf{Redseq}(c^i_1, c^a_2)]\!] & = & ([\![c^{rhs}_2]\!], [\![\mathbf{Seq}(c^i_1, c_2)]\!]) \\
 & & \quad \text{where} \quad (c^{rhs}_2, c_2) \; = \; \mathcal{C}^a_{gen}[\![c^a_2]\!] \\
\mathcal{C}^a_{gen}[\![\mathbf{Rebcond}(e^a, c^a_1, c^a_2)]\!] & = & ([\![\mathbf{Cond}(e^{rhs}, c^{rhs}_1, c^{rhs}_2)]\!], [\![\mathbf{Seq}(c, \mathbf{Seq}(c_1, c_2))]\!]) \\
 & & \quad \text{where} \quad (e^{rhs}, c) \; = \; \mathcal{E}^a_{gen}[\![e^a]\!] \\
 & & \quad \phantom{\text{where}} \quad (c^{rhs}_1, c_1) \; = \; \mathcal{C}^a_{gen}[\![c^a_1]\!] \\
 & & \quad \phantom{\text{where}} \quad (c^{rhs}_2, c_2) \; = \; \mathcal{C}^a_{gen}[\![c^a_2]\!] \\
\mathcal{C}^a_{gen}[\![\mathbf{Redcond}(e^i, c^a_1, c^a_2)]\!] & = & (\mathbf{Nterm}(s), [\![\mathbf{Cond}(e^i, \; \mathbf{Seq}(\mathbf{Rule}(s, c^{rhs}_1), c_1), \\
 & & \qquad\qquad\qquad\qquad\quad \mathbf{Seq}(\mathbf{Rule}(s, c^{rhs}_2), c_2))]\!]) \\
 & & \quad \text{where} \quad (c^{rhs}_1, c_1) \; = \; \mathcal{C}^a_{gen}[\![c^a_1]\!] \\
 & & \quad \phantom{\text{where}} \quad (c^{rhs}_2, c_2) \; = \; \mathcal{C}^a_{gen}[\![c^a_2]\!] \\
 & & \quad \phantom{\text{where}} \quad s \text{ is a fresh non-terminal} \\
\mathcal{E}^a_{gen} : Exp^a \rightarrow (Exp^{rhs} \times Com) & & \\
\mathcal{E}^a_{gen}[\![\mathbf{Eval}(e^i)]\!] & = & ([\![\mathbf{Hole}(h)]\!], [\![\mathbf{Inst}(h, e^i)]\!]) \\
 & & \text{where } h \text{ is a fresh hole} \\
\mathcal{E}^a_{gen}[\![\mathbf{Id}(e^i)]\!] & = & ([\![e^i]\!], [\![\mathbf{Nop}]\!]) \\
\mathcal{E}^a_{gen}[\![\mathbf{Rebcall}(o, e^a_1, e^a_2)]\!] & = & ([\![\mathbf{Call}(o, e^{rhs}_1, e^{rhs}_2)]\!], [\![\mathbf{Seq}(c_1, c_2)]\!]) \\
 & & \quad \text{where} \quad (e^{rhs}_1, c_1) \; = \; \mathcal{E}^a_{gen}[\![e^a_1]\!] \\
 & & \quad \phantom{\text{where}} \quad (e^{rhs}_2, c_2) \; = \; \mathcal{E}^a_{gen}[\![e^a_2]\!]
\end{array}
$$

Figure 11: Abstract interpretation of the actions

5 Related Work

Recently two approaches to run-time code generation have been reported by Engler and Proebsting [6], and by Leone and Lee [11]. These approaches include some aspects of run-time specialization and address issues related to compiling code at run time.

Engler and Proebsting's approach consists of providing the programmer with operations to construct templates manually in the intermediate representation of the LCC compiler (a form of register transfer language) [7]. Then, at run time, the operations to construct templates are executed, and a fast code generator is invoked to compile templates into binary code.

Not only is this approach error-prone because templates are written manually, but it also forces the code generation process to be overly simple because it needs to be fast (no elaborate register allocation or instruction scheduling is performed).

Leone and Lee's approach is developed for a first-order subset of a purely functional language. It is aimed at postponing certain compilation operations until run time to better optimize programs. Operations such as register allocation may be performed at run time for some program fragments. The binding-time of a given function is defined by the way it is curried.

Both approaches suffer from the fact that the run-time compiler does not have a global view of the program to be specialized, nor does it know what kind of specialized programs can be produced at run time. Consequently, run-time code generation is not performed at the level of a procedure or a basic block, it is done at the instruction level. This strategy makes it difficult to generate efficient code.

In contrast, our approach enables the compiler to process program fragments globally in that it is applied to the possible combinations of templates which can be constructed at run-time. Because the compiler processes large code fragments it is able to produce efficient code.

Many existing approaches (*e.g.,* [11, 6]) emphasize the need to perform elaborate optimizations at run time based on the fact that much more information is available then. This is a difficult challenge because of the conflicting requirements of a run-time code generator, namely, producing code at low cost to allow this process to be amortized quickly, and exploiting as much run-time information as possible to produce highly-optimized code. When the run-time code generator only focuses on the former requirement, even if the number of instructions being executed is smaller, the quality of the generated code may be such that performance is degraded. When the run-time code generator puts too much effort on optimization, the overhead may be such that the process may not be applicable to many situations.

Determining what kind of run-time code generation process is most suitable for a given situation is a difficult problem. Two important factors need to be taken into account: the overhead introduced by the run-time code generator and the frequency of execution of the code fragment to be processed at run time.

To some extent run-time specialization simplifies the issue in that it is not aimed at performing general-purpose optimizations that may or may not improve performance. Rather it is restricted to specializing programs with respect to some run-time invariants. If the program fragments to be processed offer good opportunities for specialization, the run-time specialization process will likely be amortized and performance should improve, provided the specialized code is executed many times.

Techniques to specialize object-oriented programs at run time have also been developed [1]. They are aimed at optimizing frequently executed code sections. However, these specialization techniques do not address arbitrary computations: they are limited to the optimization of certain object-oriented mechanisms such as method dispatch.

6 Conclusions and Future Directions

We have presented an approach to performing specialization at run time, based on partial evaluation technology. It consists of producing templates at compile time and transforming them so that they can be processed by a standard compiler. At run time, only minor operations need to be performed: selecting and copying templates, filling holes with run-time values, and relocating jump targets. As a result, run-time specialization is performed very efficiently and thus does not require a specialized code to be executed many times before its cost is amortized.

Our approach has been implemented for the C language, using the GNU C compiler, and is integrated in a partial evaluation system that specializes programs at compile time as well as at run time.

Future directions for this work include conducting a thorough experimentation with our C run-time specializer and performing more measurements, developing specific techniques to use run-time specialization in operating system code where specialized code may be executing when invariants become invalid, and applying the approach to different languages like ML.

Acknowlegments

The Partial Evaluation Group at Irisa and the Synthetix group at Oregon Graduate Institute provided valuable feedback on our work and detailed comments on the paper.

Thanks also due to Olivier Danvy, Pierre Jouvelot, David Keppel and Mark Leone for helpful comments on the paper and stimulating discussions.

References

[1] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[2] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, pages 88–105, Springer-Verlag, 1990.

[3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[4] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. *A Uniform Approach for Compile-Time and Run-Time Specialization.* Technical Report, University of Rennes/Inria, 1995. In preparation.

[5] C. Consel and S.C. Khoo. *On-line & Off-line Partial Evaluation: Semantic Specifications and Correctness Proofs.* Research Report, Yale University, New Haven, Connecticut, USA, 1993. Extended version. To appear in *Journal of Functional Programming.*

[6] D. R. Engler and T. A. Proebsting. DCG: an efficient, retargetable dynamic code generation system. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[7] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software - Practice and Experience*, 21(9):963–988, 1991.

[8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[9] D. Keppel, S. Eggers, and R. Henry. *A Case for Runtime Code Generation.* Technical Report, University of Washington, Seattle, Washington, 1991.

[10] D. Keppel, S. Eggers, and R. Henry. *Evaluating Runtime Compiled Value-Specific Optimizations.* Technical Report 93-11-02, University of Washington, Seattle, Washington, 1993.

[11] M. Leone and P. Lee. Lightweight run-time code generation. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, 1994.

[12] B. N. Locanthi. Fast bitblt with asm() and cpp. In *European Unix Users Group Conference Proceedings (EUUG)*, 1987.

[13] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *ACM Symposium on Operating Systems Principles*, pages 191–201, 1989.

[14] R. Pike, B. N. Locanthi, and J.F. Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985.

[15] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *ACM Symposium on Operating Systems Principles*, 1995. To appear.

[16] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *ACM Computing Systems*, 1(1):11–32, 1988.

A Correctness

In this section, the correctness proof of our approach is presented. The proof of the main theorem relies on four lemmas which relate the stores being produced by a run-time specializer and by standard interpretation. It also relates specialized expressions produced by interpretation of actions and by evaluation of a run-time specializer. Because the proofs of these lemmas are simple, they are omitted.

Definition 1 $\forall \sigma \in Store, \bar{\sigma} = \sigma[\S \mapsto \perp]$

Lemma 1 states that the evaluation of a command written in the initial language does not affect (or depend on) the program being specialized stored at location §.

Lemma 1 $(\forall c^i \in Com^i)(\forall \sigma \in Store)(\forall c^{rhs} \in Com^{rhs})$
$\mathcal{C}[\![c^i]\!]\sigma[\S \mapsto c^{rhs}] = (\mathcal{C}[\![c^i]\!]\sigma)[\S \mapsto c^{rhs}]$.

Lemma 2 stipulates that for any command written in the extended language, whether or not it is evaluated with a store defined at location § does not affect the other values contained in the store.

Lemma 2 $(\forall c \in Com)(\forall \sigma \in Store)$ $\overline{\mathcal{C}[\![c]\!]\sigma} = \mathcal{C}[\![c]\!]\bar{\sigma}$

The following two lemmas address a correctness issue regarding the expression included in assigments and conditionals. More precisely, for a given action-analyzed expression and a store, a specialized expression can be produced by interpreting the actions using $\mathcal{E}^a$. Another alternative is to evaluate the run-time specializer produced by $\mathcal{E}^a_{gen}$ for this action-analyzed expression.

Lemmas 3 and 4 state that these different evaluation strategies produce the same specialized expression and the same store modulo the value of the store at location §.

Lemma 3 $(\forall x \in Id)(\forall e^a \in Exp^a)(\forall \sigma \in Store)$
Let $(e^{rhs}, c) = \mathcal{E}^a_{gen}[\![e^a]\!]$
Then, $\sigma' = \mathcal{C}[\![c]\!]\sigma[\S \mapsto \mathbf{Assign}(x, e^{rhs})] \Rightarrow \sigma'(\S) = \mathbf{Assign}(x, \mathcal{E}^a[\![e^a]\!]\bar{\sigma}) \wedge \bar{\sigma'} = \bar{\sigma}$

The following lemma uses function $Hole$ to collect holes in rhs-terms (expressions and commands).

Lemma 4 $(\forall e^a \in Exp^a)(\forall c_1^{rhs}, c_2^{rhs} \in Com^{rhs})(\forall \sigma \in Store)$
Let $(e^{rhs}, c) = \mathcal{E}^a_{gen}[\![e^a]\!]$
Then, $Hole(e^{rhs}) \cap (Hole(c_1^{rhs}) \cup Hole(c_2^{rhs})) = \emptyset \wedge$
$\sigma' = \mathcal{C}[\![c]\!]\sigma[\S \mapsto \mathbf{Cond}(e^{rhs}, c_1^{rhs}, c_2^{rhs})] \Rightarrow \sigma'(\S) = \mathbf{Cond}(\mathcal{E}^a[\![e^a]\!]\bar{\sigma}, c_1^{rhs}, c_2^{rhs}) \wedge \bar{\sigma'} = \bar{\sigma}$

Theorem 1 $(\forall c^a \in Com^a)(\forall \sigma \in Store)$
Let $(c^{rhs}, c) = \mathcal{C}^a_{gen}[\![c^a]\!]$
Then, $\sigma' = \mathcal{C}[\![c]\!]\sigma[\S \mapsto c^{rhs}] \Rightarrow (\sigma'(\S), \bar{\sigma'}) = \mathcal{C}^a[\![c^a]\!]\bar{\sigma}$

Proof: the proof is by structural induction on c^a

- If $c^a = \mathbf{Eval}(c^i)$
  then, $\mathcal{C}^a_{gen}[\![c^a]\!] = (\mathbf{Nop}, c^i)$
  $\sigma' = \mathcal{C}[\![c^i]\!]\sigma[\S \mapsto \mathbf{Nop}]$
  By Lemma 1, $\sigma' = (\mathcal{C}[\![c^i]\!]\sigma)[\S \mapsto \mathbf{Nop}]$
  $\Rightarrow (\sigma'(\S), \bar{\sigma'}) = (\mathbf{Nop}, \mathcal{C}[\![c^i]\!]\bar{\sigma}) = \mathcal{C}^a[\![c^a]\!]\bar{\sigma}$
- If $c^a = \mathbf{Id}(c^i)$
  then, $\mathcal{C}^a_{gen}[\![c^a]\!] = (c^i, \mathbf{Nop})$
  $\sigma' = \mathcal{C}[\![\mathbf{Nop}]\!]\sigma[\S \mapsto c^i] = \sigma[\S \mapsto c^i]$
  $\Rightarrow (\sigma'(\S), \bar{\sigma'}) = (c^i, \bar{\sigma}) = \mathcal{C}^a[\![c^a]\!]\bar{\sigma}$
- If $c^a = \mathbf{Rebassign}(x, e^a)$
  then, $\mathcal{C}^a_{gen}[\![c^a]\!] = (\mathbf{Assign}(x, e^{rhs}), c_0)$ where $(e^{rhs}, c_0) = \mathcal{E}^a_{gen}[\![e^a]\!]$
  $\sigma' = \mathcal{C}[\![c_0]\!]\sigma[\S \mapsto \mathbf{Assign}(x, e^{rhs})]$
  By lemma 3, $(\sigma'(\S), \bar{\sigma'}) = (\mathbf{Assign}(x, \mathcal{E}^a[\![e^a]\!]\bar{\sigma}), \bar{\sigma})$
  $\Rightarrow (\sigma'(\S), \bar{\sigma'}) = \mathcal{C}^a[\![c^a]\!]\bar{\sigma}$

- If $c^a = \mathbf{Rebseq}(c_1^a, c_2^a)$

  then, $\mathcal{C}^a_{gen}\llbracket c^a \rrbracket = (\mathbf{Seq}(c_1^{rhs}, c_2^{rhs}), \mathbf{Seq}(c_1, c_2))$ where $\left\{ \begin{array}{lcl} (c_1^{rhs}, c_1) & = & \mathcal{C}^a_{gen}\llbracket c_1^a \rrbracket \\ (c_2^{rhs}, c_2) & = & \mathcal{C}^a_{gen}\llbracket c_2^a \rrbracket \end{array} \right.$

  $\sigma' = \mathcal{C}\llbracket \mathbf{Seq}(c_1, c_2) \rrbracket \sigma[\S \mapsto \mathbf{Seq}(c_1^{rhs}, c_2^{rhs})] = \mathcal{C}\llbracket c_2 \rrbracket (\mathcal{C}\llbracket c_1 \rrbracket \sigma[\S \mapsto \mathbf{Seq}(c_1^{rhs}, c_2^{rhs})])$

  Because only c_1(resp. c_2) can substitute non-terminals and holes introduced in c_1^{rhs} (resp. c_2^{rhs}),

  $\sigma' = \delta'[\S \mapsto \mathbf{Seq}(\delta(\S), \delta'(\S))]$ where $\left\{ \begin{array}{lcl} \delta & = & \mathcal{C}\llbracket c_1 \rrbracket \sigma[\S \mapsto c_1^{rhs}] \\ \delta' & = & \mathcal{C}\llbracket c_2 \rrbracket \delta[\S \mapsto c_2^{rhs}] \end{array} \right.$

  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = (\mathbf{Seq}(\delta(\S), \delta'(\S)), \bar{\delta}')$

  By induction, $\left\{ \begin{array}{lcl} (\delta(\S), \bar{\delta}) & = & \mathcal{C}^a\llbracket c_1^a \rrbracket \bar{\sigma} \\ (\delta'(\S), \bar{\delta}') & = & \mathcal{C}^a\llbracket c_2^a \rrbracket \bar{\delta} \end{array} \right.$

  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = \mathcal{C}^a\llbracket c^a \rrbracket \bar{\sigma}$

- If $c^a = \mathbf{Redseq}(c_1^i, c_2^a)$
  then, $\mathcal{C}^a_{gen}\llbracket c^a \rrbracket = (c_2^{rhs}, \mathbf{Seq}(c_1^i, c_2))$ where $(c_2^{rhs}, c_2) = \mathcal{C}^a_{gen}\llbracket c_2^a \rrbracket$
  $\sigma' = \mathcal{C}\llbracket \mathbf{Seq}(c_1^i, c_2) \rrbracket \sigma[\S \mapsto c_2^{rhs}] = \mathcal{C}\llbracket c_2 \rrbracket (\mathcal{C}\llbracket c_1^i \rrbracket \sigma[\S \mapsto c_2^{rhs}])$
  By lemma 1, $\sigma' = \mathcal{C}\llbracket c_2 \rrbracket (\mathcal{C}\llbracket c_1^i \rrbracket \sigma)[\S \mapsto c_2^{rhs}]$
  By induction, $(\sigma'(\S), \bar{\sigma}') = \mathcal{C}^a\llbracket c_2^a \rrbracket (\overline{\mathcal{C}\llbracket c_1^i \rrbracket \sigma})$
  By lemma 2, $(\sigma'(\S), \bar{\sigma}') = \mathcal{C}^a\llbracket c_2^a \rrbracket (\mathcal{C}\llbracket c_1^i \rrbracket \bar{\sigma})$
  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = \mathcal{C}^a\llbracket c^a \rrbracket \bar{\sigma}$.

- If $c^a = \mathbf{Rebcond}(e^a, c_1^a, c_2^a)$
  then, $\mathcal{C}^a_{gen}\llbracket c^a \rrbracket = (\mathbf{Cond}(e^{rhs}, c_1^{rhs}, c_2^{rhs}), \mathbf{Seq}(c_0, \mathbf{Seq}(c_1, c_2)))$

  where $\left\{ \begin{array}{lcl} (e^{rhs}, c_0) & = & \mathcal{E}^a_{gen}\llbracket e^a \rrbracket \\ (c_1^{rhs}, c_1) & = & \mathcal{C}^a_{gen}\llbracket c_1^a \rrbracket \\ (c_2^{rhs}, c_2) & = & \mathcal{C}^a_{gen}\llbracket c_2^a \rrbracket \end{array} \right.$

  $\sigma' = \mathcal{C}\llbracket \mathbf{Seq}(c_0, \mathbf{Seq}(c_1, c_2)) \rrbracket \sigma[\S \mapsto \mathbf{Cond}(e^{rhs}, c_1^{rhs}, c_2^{rhs})]$
  $\sigma' = \mathcal{C}\llbracket \mathbf{Seq}(c_1, c_2) \rrbracket \delta$ where $\delta = \mathcal{C}\llbracket c_0 \rrbracket \sigma[\S \mapsto \mathbf{Cond}(e^{rhs}, c_1^{rhs}, c_2^{rhs})]$
  By construction, e^{rhs}, c_1^{rhs} and c_2^{rhs} do not share holes and lemma 4 gives,
  $\delta(\S) = \mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, c_1^{rhs}, c_2^{rhs})$ and $\bar{\delta} = \bar{\sigma}$
  $\Rightarrow \delta = \sigma[\S \mapsto \mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, c_1^{rhs}, c_2^{rhs})]$
  $\Rightarrow \sigma' = \mathcal{C}\llbracket \mathbf{Seq}(c_1, c_2) \rrbracket \sigma[\S \mapsto \mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, c_1^{rhs}, c_2^{rhs})$
  $\sigma' = \mathcal{C}\llbracket c_2 \rrbracket (\mathcal{C}\llbracket c_1 \rrbracket \sigma[\S \mapsto \mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, c_1^{rhs}, c_2^{rhs})])$
  As in the case of **Rebseq**, we have,

  $\sigma' = \delta''[\S \mapsto \mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, \delta'(\S), \delta''(\S))]$ where $\left\{ \begin{array}{lcl} \delta' & = & \mathcal{C}\llbracket c_1 \rrbracket \sigma[\S \mapsto c_1^{rhs}] \\ \delta'' & = & \mathcal{C}\llbracket c_2 \rrbracket \delta'[\S \mapsto c_2^{rhs}] \end{array} \right.$

  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = (\mathbf{Cond}(\mathcal{E}^a\llbracket e^a \rrbracket \bar{\sigma}, \delta'(\S), \delta''(\S)), \bar{\delta}'')$.

  By induction, $\left\{ \begin{array}{lcl} (\delta'(\S), \bar{\delta}') & = & \mathcal{C}^a\llbracket c_1^a \rrbracket \bar{\sigma} \\ (\delta''(\S), \bar{\delta}'') & = & \mathcal{C}^a\llbracket c_2^a \rrbracket \bar{\delta}' \end{array} \right.$

  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = \mathcal{C}^a\llbracket c^a \rrbracket \bar{\sigma}$

- If $c^a = \mathbf{Redcond}(e^i, c_1^a, c_2^a)$

  then, $\mathcal{C}^a_{gen}\llbracket c^a \rrbracket = (\mathbf{Nterm}(s), \mathbf{Cond}(e^i, \mathbf{Seq}(\mathbf{Rule}(s, c_1^{rhs}), c_1), \mathbf{Seq}(\mathbf{Rule}(s, c_2^{rhs}), c_2))$

  where $\left\{ \begin{array}{lcl} (c_1^{rhs}, c_1) & = & \mathcal{C}^a_{gen}\llbracket c_1^a \rrbracket \\ (c_2^{rhs}, c_2) & = & \mathcal{C}^a_{gen}\llbracket c_2^a \rrbracket \end{array} \right.$

  $\sigma' = \mathcal{C}\llbracket \mathbf{Cond}(e^i, \mathbf{Seq}(\mathbf{Rule}(s, c_1^{rhs}), c_1), \mathbf{Seq}(\mathbf{Rule}(s, c_2^{rhs}), c_2)) \rrbracket \sigma[\S \mapsto \mathbf{Nterm}(s)]$

  $$\begin{array}{ll} \sigma' = \text{if } \mathcal{E}\llbracket e^i \rrbracket \sigma[\S \mapsto \mathbf{Nterm}(s)] & \\ \qquad \text{then} & \mathcal{C}\llbracket c_1 \rrbracket (\mathcal{C}\llbracket \mathbf{Rule}(s, c_1^{rhs}) \rrbracket \sigma[\S \mapsto \mathbf{Nterm}(s)]) \\ \qquad \text{else} & \mathcal{C}\llbracket c_2 \rrbracket (\mathcal{C}\llbracket \mathbf{Rule}(s, c_2^{rhs}) \rrbracket \sigma[\S \mapsto \mathbf{Nterm}(s)]) \end{array}$$

  $\sigma' = \text{if } \mathcal{E}\llbracket e^i \rrbracket \bar{\sigma} \text{ then } \delta_1 \text{ else } \delta_2$ where $\left\{ \begin{array}{lcl} \delta_1 & = & \mathcal{C}\llbracket c_1 \rrbracket \sigma[\S \mapsto c_1^{rhs}] \\ \delta_2 & = & \mathcal{C}\llbracket c_2 \rrbracket \sigma[\S \mapsto c_2^{rhs}] \end{array} \right.$

  By induction, $\left\{ \begin{array}{l} (\delta_1(\S), \bar{\delta}_1) = \mathcal{C}^a\llbracket c_1^a \rrbracket \bar{\sigma} \\ (\delta_2(\S), \bar{\delta}_2) = \mathcal{C}^a\llbracket c_1^a \rrbracket \bar{\sigma} \end{array} \right.$

  $\Rightarrow (\sigma'(\S), \bar{\sigma}') = \text{if } \mathcal{E}\llbracket e^i \rrbracket \bar{\sigma} \text{ then } \mathcal{C}^a\llbracket c_1^a \rrbracket \bar{\sigma} \text{ else } \mathcal{C}^a\llbracket c_2^a \rrbracket \bar{\sigma} = \mathcal{C}^a\llbracket c^a \rrbracket \bar{\sigma}$ $\square$