Chapter 1

Main Principles of Program Specialization

> Who was it that said, "Let us lean heavily on principles; they will always give way in the end"?
>
> — Edouard Herriot *Notes & Maxims*

A *program specialization* is a type of optimizing program transformation. To simplify this, we can take the following example: if we have a program, on the one hand, and a context for its execution, on the other (i.e. a partial knowledge of the data that will be input into the program at run time), we seek to create a new program, whose behavior is identical to that of the original program (same effects, same results), but which performs better because it is specialized to that particular execution context. In a broad context, *program specialization* denotes an area of computer science that brings together analytical and program transformation techniques to automatically create specialized programs.

In this chapter, we examine the main principles of program specialization. We build a cohesive framework to be used in the following chapters, which essentially discuss and develop offline specialization techniques for the C language, but whose scope is, in fact, broader than this, with the arguments being transposed or extended to C++ and Java.

Here, we focus on the motives and the stakes involved in program specialization, while attempting to maintain a fairly broad, general view. For more information about particular language cases, especially as regards binding times, the reader is referred to the publications and reference works cited in the text.

Organization of this chapter

– Section 1.1 precisely defines what a specialized program is, in terms of precomputations and semantic equivalence to the original program.

– Section 1.2 explains the advantages of specialization from the point of view of performance (in time and space).

– Section 1.3, without detailing the techniques involved in program specialization, constructs the general framework of an automated specialization process, and examines the pros and cons of this.

– Section 1.4 describes the main applications of program specialization: compiling with an interpreter (and more generally getting rid of layers of interpretation), changing an interpreter into a compiler, and creating a compiler generator.

– Section 1.5 distinguishes two specialization times (compile time and runtime) and examines the uses to which the resulting specialized programs could be put (as a specialization server and a specialization cache).

– Section 1.6, finally, discusses questions relating to whether or not specialization is profitable, particularly in terms of time and space saved.

1.1. Specialized program

In this section, we define the most commonplace concepts associated with program specialization, with the question of automatic production of a specialized program being dealt with in subsequent sections.

1.1.1. *Program specialization*

Take $L : Prog \rightarrow Input \rightarrow Output$, a programming language. A version of a program $p \in Prog$, specialized to a partial input $in_{\in} Input$, is a program $p_{in_{\mathtt{s}}} \in Prog$ such that for any complete input $in \in Input$ that can be broken down into subinputs $(in_{\mathtt{s}}, in_{\mathtt{d}}) = in$, we have:

$$[\![p_{in_{\mathtt{s}}}]\!]_L \; in_{\mathtt{d}} \quad = \quad [\![p]\!]_L \; (in_{\mathtt{s}}, in_{\mathtt{d}}) \qquad [1.1]$$

The $p_{in_{\mathtt{s}}}$ program is also known as a *specialized program*. In certain ways, it is equivalent to the program p for the input values in that make up the partial input $in_{\mathtt{s}}$ – equivalent, in the sense that it produces the same output – that is, it has the same effects and provides the same results:

$$[\![p_{in_{\mathtt{s}}}]\!]_L \; in_{\mathtt{d}} \quad = \quad [\![p]\!]_L \; (in_{\mathtt{s}}, in_{\mathtt{d}}) \quad = \quad [\![p]\!]_L \; in \quad = \quad out \qquad [1.2]$$

If we are only interested in the input in that forms part of the partial input $in_{\mathtt{s}}$, then the specialized program $p_{in_{\mathtt{s}}}$ can, in a manner of speaking, be "substituted" for program p. Strictly speaking, the input channels of $p_{in_{\mathtt{s}}}$ are included in those of p, and the programs p and $p_{in_{\mathtt{s}}}$ are only indirectly comparable from a semantic point of view (see section 1.1.6). The above definitions are represented diagrammatically in Figure 1.1.

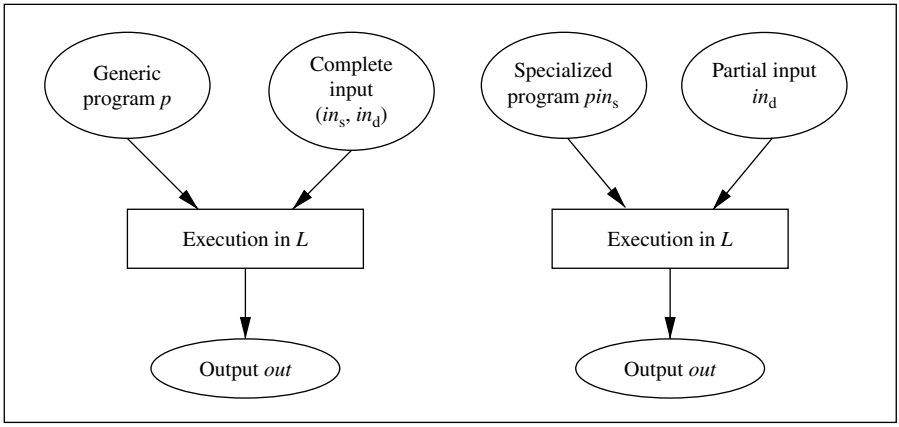

Figure 1.1. *Generic program and specialized program*

On the other hand, program p has more input channels than $p_{in_{\mathtt{s}}}$ – it can operate in more execution contexts. That is why, by contrast to the specialized program $p_{in_{\mathtt{s}}}$, p is referred to as a *generic program*.

NOTE 1.1. – This definition is purely semantic – it does not imply any link between the generic and the specialized programs other than the inclusion of identical input to yield identical output. In practice, a specialized program does not simply materialize, but rather it is obtained by transforming a generic program (see section 1.3). Also, as so often happens with nominalizations of verbs, the term "specialization" denotes both the action (of transforming a generic program p into a specialized program $p_{in_{\mathtt{s}}}$) and the result of that action (a specialized program $p_{in_{\mathtt{s}}}$). However, in this section, we discuss only the actual nature of the *specialized code*.

The known partial input $in_{\mathtt{s}}$ to which the program is specialized is termed *static input*. The additional partial input $in_{\mathtt{d}}$ is called *dynamic input*. This terminology illustrates the fact that, for a specialized program $p_{in_{\mathtt{s}}}$, the static input $in_{\mathtt{s}}$ is fixed, whereas the dynamic input $in_{\mathtt{d}}$ can still vary, as it remains unknown up until the moment of execution. The input values of $in_{\mathtt{s}}$ are called *specializing values*.

According to this definition, in is not restricted to just the input that is compatible with p, nor is $in_{\mathtt{d}}$ restricted just to the input that is compatible with $p_{in_{\mathtt{s}}}$

(see section A.2.9). In particular, the semantic terms $[\![p_{in_s}]\!]_L\ in_d$ and $[\![p]\!]_L\ (in_s, in_d)$ may be equal to a defined error in output, err. Consequently, if the partial input in_s is incompatible with p, then p_{in_s} is a program whose execution systematically produces an output with a definite error err. Also, remember that here we assume a program to be complete and not to interact with external objects (see section A.2.10), hypotheses that will be raised in Chapter 8.

We do not have to specialize all of a program; we can specialize only a portion of it – in practice, one or more subprograms (see section 9.1). However, to avoid ambiguity, hereafter we will not speak of specializing subprograms, but of specializing programs. Similarly, we will use the term *specialized program* to refer to a program–one of whose subprograms is specialized. When we use a subprogram accompanied by the functions it calls upon (to an arbitrary call depth), we call this the *specialization point of entry*.

In sections 3.3.3 and 6.6, we will look at a variant of this definition of specialization in which an input in_{sd} to the program may be considered to be both static and dynamic. The idea is that it serves *simultaneously* to generate the specialized code $p_{in_s, in_{sd}}$ and execute it $[\![p_{in_s, in_{sd}}]\!]\ (in_{sd}, in_d)$. Notably, this variant enables us to look at cases where a piece of information cannot easily be integrated into a program's code.

Examples of specialization

Take, for example, "program" C shown in Figure 1.2, which is deliberately very simple. It is a three-argument function that calculates a scalar product[1]. A specialization of that function `dotprod`, relating to the partial input `size` $=$ 3, is given in Figure 1.3. The specialized function `dotprod_size3` is such that whatever the values of x and y, a call to `dotprod_size3(`x`,`y`)` is semantically equivalent to a call to the generic function `dotprod(3,`x`,`y`)`: the same value is returned and the effects on the memory are identical (in this case, none).

Note that the value of the static parameter `size` is fixed; for example, we cannot calculate the equivalent of `dotprod(2,`x`,`y`)` using the function `dotprod_size3`. For this, we need another specialized function, namely `dotprod_size2` (see Figure 1.4).

Figure 1.5 gives the example of `dotprod_size3_u50m1`, another specialization of the function `dotprod`, this time relating to the partial input values `size` $=$ `3` and `u` $=$ `{5,0,-1}` (a notation which we will use to represent the value of a table, which is similar to the syntax used in C to initialize a table when declaring

1 It is fairly rare for a scalar product to operate on integers. We have made it so here for educational purposes so that it is easier to speak of the complementary nature of certain optimizations of the compiler, without having to deal with issues about representing floating values, which would distract us from our main topic.

it). However, a call to `dotprod_size3_u50m1(y)` is semantically equivalent to a call to the generic function `dotprod(3,u,y)` where $u = \{5, 0, -1\}$.

```
int dotprod(int size, int u[], int v[])
{
  int rslt = 0;
  int i;
  for (i = 0; i < size; i++)
    rslt += u[i] * v[i];
  return rslt;
}
```

Figure 1.2. *Generic* `dotprod` *function*

```
int dotprod_size3(int u[], int v[])
{
  return u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
}
```

Figure 1.3. *Specialized* `dotprod` *function for* `size` $= 3$

```
int dotprod_size2(int u[], int v[])
{
  return u[0]*v[0] + u[1]*v[1];
}
```

Figure 1.4. *Specialized* `dotprod` *function for* `size` $= 2$

```
int dotprod_size3_u50m1(int v[])
{
  return 5*v[0] - v[2];
}
```

Figure 1.5. *Specialized* `dotprod` *function for* `size` $= 3$ *and* `u` $= \{5, 0, -1\}$

We might also note that the generic function `dotprod` and the specialized functions `dotprod_size3` and `dotprod_size3_u50m1` all have different input channels. However, the functions `dotprod_size3` and `dotprod_size2` have the same channels.

The particular advantage to specialization comes from the fact that specialized functions such as `dotprod_size2`, `dotprod_size3`, and

`dotprod_size3_u50m1` are relatively simpler than the generic function `dotprod`, in the sense (informally) that they perform fewer computations for a given dynamic input. In fact, they really do perform better (see section 1.2).

Of course, specialization is not restricted to such simple programs. *A priori*, it relates to programs of any degree of complexity and any size, which may redefine all the constructs in the language. For instance, in C, this includes `while` and `goto` loops, structures and links, pointer arithmetic and dereferences, direct and indirect function calls (via a pointer), dynamic memory allocation, etc.

1.1.2. *Context of specialization*

We use the term *specialization conformation* of a program or subprogram to denote the specification of the static input channels to which it is specialized. For instance, for the function `dotprod`, the specialization conformation corresponding to `dotprod_size3` is $\{\texttt{size}\}$ and that corresponding to `dotprod_size3_u50m1` is $\{\texttt{size}, \texttt{u}\}$. (This notion will be developed, in particular, in the context of the so-called offline specialization, along with the concept of binding time, see section 3.1.1).

We use the term *specialization context* of a program or subprogram to denote an abstraction of an execution context (see section A.4.1.3) – an abstraction that includes all the information relating to which the program or subprogram is specialized. In particular, a specialization context specifies not only the static input channels but also the values. Specialization contexts for the `dotprod` function are, e.g. "$\texttt{size} = 2$", "$\texttt{size} = 3$", or "$\texttt{size} = 3$ and $\texttt{u} = \{5, 0, -1\}$". Rather than specialization context, we also speak of *specialized execution context* or *case of (specialized) execution*, particularly when we wish to speak of the execution context at the moment the specialized function is called (see Chapter 9).

Conversely, an execution context is *compatible* with a specialization context if it is a concretization – i.e. an overspecification – thereof (see also section A.4.1.3). For any execution context compatible with a specialization context, the corresponding specialized program may be substituted for the generic program. Thus, the execution context "`size = 3,u= {1,2,3}` and `u= {4,5,6}`" of `dotprod` is compatible with the specialization context "`size = 3`", and the specialized function call `dotprod_size3(u,v)` may substitute `dotprod(3,u,v)`.

A function-specialization context might include, e.g. arguments from that function, global variables, and memory locations in the heap. However, a specialization context does not contain only a known value for each of the static inputs. In the case of a language with pointers like C, it may also contain alias information about the variables in the program at the moment the function to be specialized is called. More

generally, we need to model the context in which the function to be specialized is called (see section 8.4). The specialization context may also include a modelization of the effects of external functions (if there are any) called by the function to be specialized (see section 8.5). Collectively, these data constitute the *specialization parameters*.

```
void translate(char *from, char *to, char *str) {
  while(*str != '\0') {
    int i;
    for(i = 0; from[i] != '\0'; i++)
      if (*str == from[i]) {
        *str = to[i];
        break;
      }
    str++;
  }
}
```

Figure 1.6. *Generic* `translate` *function*

```
void translate_fromabc(char *to, char *str) {
  while(*str != '\0') {
    if (*str == 'a')
      *str = to[0];
    else if (*str == 'b')
      *str = to[1];
    else if (*str == 'c')
      *str = to[2];
    str++;
  }
}
```

Figure 1.7. *Specialized* `translate` *function for* `from = "abc"`

By way of illustration, consider the `translate` function in Figure 1.6, which reads the string `str` and replaces every character there that is present in the string `from`, at a position i, with the character located at the same position i in the string `to`. For instance, if the value of the variable s is `"abcd"` (i.e. if its value is a pointer to an area of memory containing the series of characters `'a'`, `'b'`, 'c', `'d'`, `'\0'`), then following a call to `translate("abc","bcd",s)`, the value of `s` is `"bcdd"`. A specialization of `translate` to the partial input `from` = "abc" is given in Figure 1.7.

We might believe that if the value of the variable `x` is `"abc"`, then `translate(x,y,z)` always behaves in the same way as `translate_fromabc(y,z)`, and therefore the two forms are mutually interchangeable in any execution context. This is absolutely not so. Indeed, following the call `translate(x,"bcd",x)`, the value of `x` is "`bdd`"; on the other hand, after `translate_fromabc("bcd",x)`, the value of `x` is "`bcd`".

The difference in behavior arises from the fact that the memory associated with the `from` string may vary when `translate` is called (by the assignments `*str = to[i]`), whereas it is more or less constant in the specialized version `translate_fromabc`. However, the two forms `translate(x,y,z)` and `translate_fromabc(y,z)`, always behave in the same way and are, therefore, interchangeable in any execution context where the value of `x` is `"abc"` and where `z` is not an alias of `x` (or, more precisely, if `z` does not point to the same area of memory as does `x`).

In a specialization context such that the parameter `from` is declared as not being an alias of the parameter `str`, the function `translate_fromabc` may be considered to be a specialization of `translate`. On the other hand, with a specialization context with no restriction on the alias relation of the arguments, i.e. for all possible execution contexts, the function `translate_fromabc` cannot be considered a specialization of `translate`. (In this particular example, the absence of alias between `from` and `str` is only one condition sufficing for the substitutability of the specialized function. There are other sufficing conditions, specific to `translate`, such as the absence of repeated characters in `from` and common characters between `from` and `to`).

1.1.3. *Specialization of a fragment of program*

Certain specializers have a *granularity of specialization* which is finer than that of the function: they facilitate the *specialization of program fragments* even within a function.

This does not significantly alter the problem. Like a function, a code fragment to be specialized has an *entry point* and *exit points* that are clearly identified. It generally corresponds to a basic block or to a connected set of basic blocks. Again in the same way as a function, the code fragment has its *input channels* and *values* and its *output channels* and *values*. However, unlike a function, there is no call stack at the entry point of such a code fragment. Nevertheless, this entry point may be the target of several branches in the function to which it belongs, and hence it may have different *execution contexts*. Even if it is only the target of one branch, a code fragment may be in a loop and therefore be executed in different execution contexts. In the same way as for the specialization of functions, there is then also the notion of the

specialization context, specific to the case of the program fragments, but similar to the specialization context defined when the granularity of specialization is the function (see section 1.1.2).

In what follows, we will speak almost exclusively of the specialization of functions. However, most of what is said can be transposed to the specialization of program fragments.

1.1.4. *Partial computations*

In terms of its interface, a specialized program expresses a sort of partial application (with respect to (w.r.t.) λ-calculus); in terms of its contents, it expresses a partial execution.

1.1.4.1. *Partial application*

A specialized program p_{in_s} may be viewed as a *curried* and partially applied form of p. Indeed, we have:

$$\begin{aligned} [\![p_{in_s}]\!] &= \lambda x_2 \,.\, [\![p_{in_s}]\!]\, x_2 \\ &= \lambda x_2 \,.\, [\![p]\!]\, (in_s, x_2) \quad = \quad (\lambda x_1 . \lambda x_2 \,.\, [\![p]\!]\, (x_1, x_2))\; in_s \end{aligned} \qquad [1.3]$$

In this book, we will use λ-terms only as an algebraic mathematical notation to define a function easily, not as a genuinely manipulatable term.

In a functional language such as Scheme or ML, *partial application* is a construct that is peculiar to the language, which produces a new functional value (generally implemented as a *closure*). The equivalent here would be to form the partial application $p(in_s)$, which could then be applied to $in_d : (p(in_s))(in_d)$. It is on this principle that Fabius is based [LEE 96] – a specializer of a subset of pure, first-order ML (with no side effect).

However, a difference between the partial application of functional languages and this type of partial application carried out by specialization is that the former applies only to the first syntactical argument (x_1 in $\lambda x_1 . \lambda x_2 . exp$) whereas the latter may apply to any input in the function (including x_2). Another (major) difference is that partial application in specialization is coupled with immediate partial execution; we do not wait for all the arguments to be supplied before beginning to execute the code associated with the function.

1.1.4.2. *Partial execution*

The general idea largely underlying program specialization is that, because the static input in_s no longer varies for the specialized program p_{in_s}, operations relating

only to that input $in_{\mathtt{s}}$ may even at that point be *preexecuted* (or *precomputed*). A specialized program $p_{in_{\mathtt{s}}}$ may therefore result from a sort of symbolic *partial execution* of p on the static input $in_{\mathtt{s}}$ (Also see section 2.1.1 for the notion of *partial evaluation* and section 2.4 for other forms of specialization).

The pre-executable terms and code fragments of p for a given partial input $in_{\mathtt{s}}$ are qualified as *static*; the others, which are not pre-executable because of their dependency on the value of the complementary input $in_{\mathtt{d}}$, are known as *dynamic*. We also speak of *static computation* for pre-executable or pre-executed computation.

```
int dotprod(int size, int u[], int v[])
{
  int rslt = 0;
  int i;
  for (i = 0; i < size; i++)
    rslt += u[i] * v[i];
  return rslt;
}                                        // Bold: pre-executable fragments
```

Figure 1.8. *Pre-executable fragments of* `dotprod` *when* `size` *is known*

```
int dotprod(int size, int u[], int v[])
{
  int rslt = 0;
  int i;
  for (i = 0; i < size; i++)
    rslt += u[i] * v[i];
  return rslt;
}                                        // Bold: pre-executable fragments
```

Figure 1.9. *Pre-executable fragments of* `dotprod` *when* `size` *and* `u` *are known*

Ideally, in a specialized program $p_{in_{\mathtt{s}}}$, only the operations that depend on the dynamic input $in_{\mathtt{d}}$ remain to be executed in order to produce the same result as $[\![p]\!]\ (in_{\mathtt{s}}, in_{\mathtt{d}})$. Hence, we can consider the program $p_{in_{\mathtt{s}}}$ to be more efficient (faster) than the program p because it has fewer computations to carry out.

Consider, for example, the `dotprod` function shown in Figure 1.8. All operations relating to the input `size` have been put in bold. These operations can be preexecuted as soon as the value of `size` is known. The first line of the function (the initialization of `rslt`), which is independent of the arguments, is also pre-executable. On the other hand, the operations not shown in bold cannot be pre-computed while the values of `u`

and v are unknown. For instance, if the value of size is equal to 3, pre-execution can generate a function dotprod_size3 such as that shown in Figure 1.3.

1.1.4.3. *Specialization without static input*

A particular case of specialization is when the partial static input $in_{\mathtt{s}}$ of a program p is null, i.e. when the ensemble of the static input channels is null. In this case, the dynamic inputs, complementary to $in_{\mathtt{s}} = \varnothing$, are the standard inputs of p, and there is semantic equivalence $[\![p_{\varnothing}]\!] = [\![p]\!]$. However, there is not necessarily equality $p_{\varnothing} = p$ (except in the case of a trivial specialization, see below), because specialization can still be carried out on the body of p.

This *specialization without static input* makes sense; it corresponds to a case where we can exploit the constant values present in the code as though they were static inputs, in order to carry out precomputations. In fact, in order to be precomputable, a term may either depend on static inputs, or depend simply on constants within the program.

In reality, it is more correct to define a term as *non-precomputable* if it depends on dynamic inputs, and *precomputable* otherwise – i.e. if it does not depend on dynamic inputs, but may depend on constants or static inputs. However, for specialization techniques such as deforestation and supercompilation (see section 2.4) that deal more with expressions (patterns) than with values, this notion of dependency is not applicable.

1.1.5. *Range of specializations*

Creating a specialized program $p_{in_{\mathtt{s}}}$ is a complex operation because we must know how to "unravel" the computations programmed in p in order to distinguish the precomputable terms, particularly those relating only to $in_{\mathtt{s}}$, which can be carried out in advance, from those which are non-precomputable as they may also relate to $in_{\mathtt{d}}$, which therefore cannot usually be altered. However, it is not inconceivable for all the computations relating to $in_{\mathtt{s}}$ to be precomputed in the code of $p_{in_{\mathtt{s}}}$ so that $p_{in_{\mathtt{s}}}$ is qualified as a "specialized program". In fact, specialization is not unique, even when the specialization context $in_{\mathtt{s}}$ is fixed; a whole range of *variants* are envisageable.

Indeed, given a program p and a partial input $in_{\mathtt{s}}$, the general problem involves finding programs p' that satisfy the equation $[\![p']\!]\ in_{\mathtt{d}} = [\![p]\!]\ (in_{\mathtt{s}}, in_{\mathtt{d}})$ for every partial complementary input $in_{\mathtt{d}}$. This problem may have an infinite number of solutions. In practice, we seek a means of automatically constructing solutions to this problem by transforming programs (see section 1.3). We also seek solution programs that are as high performance as possible (see section 1.2), i.e. generally that integrate a maximum of precomputations concerning $in_{\mathtt{s}}$ [MAR 94].

```
int dotprod_size3_unroll(int u[], int v[])
{
  int rslt = 0;
  int i;
  i = 0;
  rslt += u[i] * v[i];
  i = 1;
  rslt += u[i] * v[i];
  i = 2;
  rslt += u[i] * v[i];
  return rslt;
}
```

Figure 1.10. *Specialized* `dotprod` *function (loop unrolling only)*

```
int dotprod_size3_propag(int u[], int v[])
{
  int rslt = 0;
  rslt += u[0] * v[0];
  rslt += u[1] * v[1];
  rslt += u[2] * v[2];
  return rslt;
}
```

Figure 1.11. *Specialized* `dotprod` *function (unrolling and propagated indices)*

1.1.5.1. *Different precomputations*

The main source of variants of specialization arises from the number of precomputations carried out. Figures 1.10 and 1.11 give different examples of specializations of the function `dotprod` to the partial input `size` $=$ 3, specializations other than `dotprod_size3` (see Figure 1.3) because they correspond to different precomputations.

In the function `dotprod_size3_unroll`, all the computations that affect the loop variable `i` (i.e. each increment) have been carried out, as have all branching decisions relating to that variable (to repeat or exit the loop), but the references to `i` in table readings have not been precomputed.

In the function `dotprod_size3_propag`, all references to the loop variable `i` have been replaced by a constant (in table keys) and the affectations of `i`, which are now useless, have been deleted.

1.1.5.2. *Specialization opportunity*

A specialization opportunity is a fragment of code in a program that may be precomputed by specialization in a certain execution context (Note: this is when the fragment and the context occur simultaneously). In practice, the title of specialization opportunity only deserves to be applied to fragments and contexts that are likely to improve the program's performance (see section 1.2), i.e. which correspond to substantial precomputations, either in terms of reducing size or in terms of computation time (by enumerating the possible multiple executions of the fragment).

In standard use, the code fragments considered as specialization opportunities are generally at the level of the function's granularity – hence, we are interested in the number of precomputations in a certain function for certain static inputs. It is often in this guise that the question of *specialization opportunity-searching* is raised: given a program and a context for its execution, we search in the functions of that program and their execution contexts for instances that constitute good specialization opportunities, i.e. specialization opportunities that would lead to significant gains in terms of performance (see sections 9.1.2 and 12.3.2).

1.1.5.3. *Degree of specialization*

The *degree of specialization* of a function is a measurement (usually informal) of the quantity of precomputations carried out during the specialization of that function. The choice of static input channels has an impact on the degree of specialization, as does a specializer's capacity to identify the terms to be precomputed, in the context of automated specialization.

This notion is related to that of a specialization opportunity. The degree of specialization is, in a manner of speaking, an absolute measure (few or many precomputations) whereas the specialization opportunity is a relative measurement (few or many precomputations in relation to the total number possible). It should also be noted that a high degree of specialization for a code fragment implies the specialization of a large number, or most, of its specialization opportunities. On the other hand, a low degree of specialization may be due to the intrinsic lack of specialization opportunities in the function, or due to a shoddy specializer that cannot identify the specialization opportunities or manage to exploit them.

1.1.5.4. *Trivial specialization*

The extreme case is when no precomputations at all are carried out. Consider the *specialized trivial program* $\mathit{triv}_{p,in_\mathtt{s}}$, defined as the program that has the same input channels $x_\mathtt{d}$ as a dynamic input $in_\mathtt{d}$ that is complementary of $in_\mathtt{s}$, and the body of which, is constituted by the call to p on the whole input $(in_\mathtt{s}, x_\mathtt{d})$. Informally, we have $\mathit{triv}_{p,in_\mathtt{s}} = \lambda x_\mathtt{d}.p(in_\mathtt{s}, x_\mathtt{d})$. All sufficiently generalist languages allow this

kind of construction, but the form this may take in practice varies greatly from one language to another, and may even show different forms within the same language. An example applied to the `dotprod` function, for the partial input `size` $=$ 3, is shown in Figure 1.12.

```
int dotprod_size3_triv(int u[], int v[])
{
  return dotprod(3, u, v);
}
```

Figure 1.12. *Trivial specialization of* `dotprod` *for* `size` $= 3$

```
int dotprod_size3_triv'(int u[], int v[])
{
  int size = 3;
  int rslt = 0;
  int i;
  for (i = 0; i < size; i++)
    rslt += u[i] * v[i];
  return rslt;
}
```

Figure 1.13. *Trivial specialization of* `dotprod` *for* `size` $= 3$ *(variant)*

The program $\mathit{triv}_{p,in_\mathtt{s}}$ is a *trivial specialization* of p to $in_\mathtt{s}$: it is a program that, in a manner of speaking, "waits" to be provided with the partial input $in_\mathtt{d}$ before executing the generic program p on the whole input $(in_\mathtt{s}, in_\mathtt{d})$, in the manner of a partial functional application (see section 1.1.4.1). To a certain extent, this is the "degree zero" of specialization because no precomputation is carried out either on the inputs $in_\mathtt{s}$ or on the possible constant values in p. Only its semantics $[\![\mathit{triv}'_{p,in_\mathtt{s}}]\!]\ in_\mathtt{d} = [\![p]\!]\ (in_\mathtt{s}, in_\mathtt{d})$ is in fact a specialization of p.

A variant of trivial specialization consists of defining a program $\mathit{triv}'_{p,in_\mathtt{s}}$ that also has the same input channels $x_\mathtt{d}$ as a dynamic input $in_\mathtt{d}$, complementary to $in_\mathtt{s}$, and whose body is defined as an affectation of the value of $in_\mathtt{s}$ on the corresponding formal parameters, followed by the code of p. Informally, we have $\mathit{triv}'_{p,in_\mathtt{s}} = \lambda x_\mathtt{d}.\{x_\mathtt{s} := in_\mathtt{s};\ p\}$. Most generalist languages also allow this type of construction, with different forms for the affectation of the variable. We can also see this variant as the result of inlining of the previous case (see section 2.1.3). An example of a trivial specialization of this type is given in Figure 1.13.

The advantage of trivial specialization is overarchingly theoretical[2]. However, it may also sometimes have a practical advantage, particularly when providing the program with the input values is slower than including them beforehand in the program itself, e.g. if they have to be read on disk and converted in to an internal processing format[3].

1.1.6. *Equivalence between the specialized program and the generic program*

As mentioned in the previous section, the specialized program has fewer inputs than the generic program (except in the particular case of specialization without static input, see section 1.1.4.3). Their definition domains are therefore not comparable. To say that a specialized program retains the semantics of generic program, the general definition of program equivalence (see section A.7.2) needs to be adapted.

Let $L : p \rightarrow \mathit{Input} \rightarrow \mathit{Output}$ be a programming language, p a program written in L, $in_{\mathtt{s}}$ a partial entry for *Input*, and $p_{in_{\mathtt{s}}}$ a specialization of p to $in_{\mathtt{s}}$. It only makes sense to compare the generic program p to the specialized program $p_{in_{\mathtt{s}}}$ on the input $in_{\mathtt{d}}$ complementary to $in_{\mathtt{s}}$. We then say that the specialized program $p_{in_{\mathtt{s}}}$ is *strictly* (respectively *lazily*) *equivalent w.r.t. specialization* to the generic program p if and only if (iff) the following respective conditions are satisfied[4].

$$p \overset{\triangleright}{\equiv}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad [\![p]\!] \circ (\lambda in_{\mathtt{d}} \,.\, (in_{\mathtt{s}}, in_{\mathtt{d}})) = [\![p_{in_{\mathtt{s}}}]\!] \qquad [1.4]$$

$$p \overset{\triangleright}{\sqsubseteq}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad [\![p]\!] \circ (\lambda in_{\mathtt{d}} \,.\, (in_{\mathtt{s}}, in_{\mathtt{d}})) = [\![p_{in_{\mathtt{s}}}]\!]_{|\,\{in_{\mathtt{d}} \in \mathit{Input} \,\mid\, (in_{\mathtt{s}}, in_{\mathtt{d}}) \in \mathit{Dom}(p)\}} \qquad [1.5]$$

---

2 Kleene's iteration theorem (notated theorem S_n^m) [KLE 52] indirectly defines a specializer for the partial recursive functions, a specializer that produces trivial specializations (the theorem, on the other hand, is not trivial).

3 The program xphoon, which sets the root window X Window to a picture of the moon with its different phases, illustrates this scenario. This specific program was written, at the time in 1988, because the program for setting the root window (`xsetroot`) was too slow: it took approximately 15 seconds to load a bitmap. By including that image in the form of information in the program, compiled with the rest of the code, the execution time fell to less than 1 second (the circumstances of xphoon's birth are described on its ReadMe page).

4 In a certain sense, the term "equivalence" may be considered misleading because the programs p and $p_{in_{\mathtt{s}}}$ do not have the same input channels. In fact, the relations $\overset{\triangleright}{\equiv}_{in_{\mathtt{s}}}$ and $\overset{\triangleright}{\sqsubseteq}_{in_{\mathtt{s}}}$ do not admit the same types of programs on the left and on the right of the relation sign. The symbol $\triangleright$ above these signs is a simple reminder.

Or, if we want to highlight the terminating executions:

$$\mathit{Dom}(p)_{\ominus in_{\mathtt{s}}} \stackrel{\text{def}}{=} \{in_{\mathtt{d}} \in \mathit{Input} \mid (in_{\mathtt{s}}, in_{\mathtt{d}}) \in \mathit{Dom}(p)\} \quad [1.6]$$

$$p \stackrel{\triangleright}{\equiv}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad \begin{cases} \mathit{Dom}(p)_{\ominus in_{\mathtt{s}}} = \mathit{Dom}(p_{in_{\mathtt{s}}}) \\ \forall\, in_{\mathtt{d}} \in \mathit{Dom}(p)_{\ominus in_{\mathtt{s}}} \;\; [\![p]\!]\,(in_{\mathtt{s}}, in_{\mathtt{d}}) = [\![p_{in_{\mathtt{s}}}]\!]\; in_{\mathtt{d}} \end{cases} \quad [1.7]$$

$$p \stackrel{\triangleright}{\sqsubseteq}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad \begin{cases} \mathit{Dom}(p)_{\ominus in_{\mathtt{s}}} \subset \mathit{Dom}(p_{in_{\mathtt{s}}}) \\ \forall\, in_{\mathtt{d}} \in \mathit{Dom}(p)_{\ominus in_{\mathtt{s}}} \;\; [\![p]\!]\,(in_{\mathtt{s}}, in_{\mathtt{d}}) = [\![p_{in_{\mathtt{s}}}]\!]\; in_{\mathtt{d}} \end{cases} \quad [1.8]$$

Alternatively, we can say that a specialized program $p_{in_{\mathtt{s}}}$ is *strictly* (respectively *lazily*) *equivalent w.r.t. specialization* to a generic program p iff it is strictly (respectively lazily) equivalent to the trivial specialization $\mathit{triv}_{p,in_{\mathtt{s}}}$ (see section 1.1.5) in the normal sense of program equivalence (see section A.7.2). In other words:

$$p \stackrel{\triangleright}{\equiv}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad \mathit{triv}_{p,in_{\mathtt{s}}} \equiv p_{in_{\mathtt{s}}} \quad [1.9]$$

$$p \stackrel{\triangleright}{\sqsubseteq}_{in_{\mathtt{s}}} p_{in_{\mathtt{s}}} \quad \text{iff} \quad \mathit{triv}_{p,in_{\mathtt{s}}} \sqsubseteq p_{in_{\mathtt{s}}} \quad [1.10]$$

This definition assumes that it is always possible to construct a trivial specialized program.

It should be noted that actually these definitions relate only to the "final interface" of the specialized program. In the course of transformation, i.e. between successive stages of transformation, it is the ordinary definitions of strict and lazy equivalence (see section A.7.2) that apply. In other words, it may be considered that we have a first stage of equivalence w.r.t. specialization, as defined above, in order to go from p to $\mathit{triv}_{p,in_{\mathtt{s}}}$, and then stages of equivalence in the normal sense to yield $p_{in_{\mathtt{s}}}$.

1.2. Specializing to improve performance

A specialized program does fewer things than a generic program, but it does them better. By restricting a program to a given specific use, program specialization allows us to improve its performance (see section A.5) for that particular use. It usually reduces execution time, and sometimes the size of the code. It may also reduce power consumption [CHU 01].

1.2.1. *Execution time*

If it has fewer computations to perform than the generic program p, we might expect that the specialized program p_{in_s} will be faster, i.e. it will have a shorter execution time (see section A.5.1):

$$\mathit{time}[\mathit{exec}\ p_{in_s}\, in_d] \overset{?}{<} \mathit{time}[\mathit{exec}\ p(in_s, in_d)] \qquad [1.11]$$

This indeed is the case for the specialized function `dotprod_size3`: all the computations in `dotprod` that depended on the parameter `size` have already been carried out; only the other computations, which depend on the two parameters `u` and `v`, remain. For whatever arguments x and y, the execution time of the specialized function `dotprod_size3(x,y)` is less than the execution time of the generic function `dotprod(3,x,y)` in any reasonable execution model.

However, as we will see below (see section 1.2.3), non-monotonous specificities of the execution machines mean that we cannot always guarantee that a specialized program will have a better execution time. This is not peculiar to program specialization; certain optimizations of *optimistic compilation* may accelerate execution in most cases but hinder it in particular situations.

In addition, note that because the specialized program carries out fewer computations than the generic program, it is also possible that it will allocate and free up less memory. In languages with garbage collection (GC), or implementations of languages that have been equipped with GC, the operations of dynamic memory management of a specialized program may also be reduced. All this contributes to a reduction in execution time.

1.2.2. *Memory space*

Performance improvement relates also to the size of the program, and in particular its static size (see section A.5.2): if it has fewer computations to carry out than the generic program p, we would expect the specialized program p_{in_s} to be smaller:

$$\mathit{size}[p_{in_s}] \overset{?}{<} \mathit{size}[p] \qquad [1.12]$$

In fact, `dotprod_size3` is smaller than `dotprod`, whatever the measuring unit used: number of lines of code, number of characters, or number of bytes of the compiled program (with various levels of optimization and on different material architectures).

However, because two different uses of the same code fragment can be specialized differently, specialization may also duplicate code fairly precisely. This is the case

with the addition/multiplication "`+ u[`i`] * v[`i`]`", which is reproduced three times here but could be reproduced more times. In fact, Figure 1.14 gives an example of the specialization of `dotprod` to the partial input `size` $=$ `10000`, where the increase in size is manifest. Hence, specialization does not always guarantee reduced program sizes.

```
int dotprod_size10000(int u[], int v[])
{
  return u[0]*v[0] +
         u[1]*v[1] +
         u[2]*v[2] +
            ...    +
         u[9997]*v[9997] +
         u[9998]*v[9998] +
         u[9999]*v[9999];
}
```

Figure 1.14. *Specialization of* `dotprod` *for* `size` $=$ `10000`

The situation is the same in functional or logic programming with recursive inlining of functions or predicates. In an incorrect use of language (due to the tropism here orientated toward C), we mainly speak of loop unrolling. However, what is said on this topic is generally applicable to recursive inlining as well.

Finally, we might also expect the dynamic size of a specialized program to be smaller than that of a generic program:

$$size[\text{exec } p_{in_{\mathtt{s}}}\ (in_{\mathtt{d}})] \overset{?}{<} size[\text{exec } p\ (in_{\mathtt{s}}, in_{\mathtt{d}})] \qquad [1.13]$$

Indeed, when fewer computations are carried out, we also avoid certain dynamic memory allocations that are only useful during precomputation. Thus, the allocations serving only for the intermediary computations can be eliminated. Only the memory space needed to store the results of these precomputations remains.

1.2.3. *Effect of the compiler*

The execution model (or execution platform) and the possible optimizations of it (see section A.5.3) are very important in comparing the performance of a generic program and a specialized program, and also between different specialized programs. In reality, in the examples below, the difference in performance is determined by the C compiler used, and by the optimizations available to it.

For instance, with the compiler gcc with no optimization (i.e. with the option `-O0`), the specialized functions defined above – `dotprod_size3_unroll`, `dotprod_size3_propag`, and `dotprod_size3` – are of noticeably different sizes. On the other hand, with a reasonable level of optimization (option `-O1`), these functions are all the same size and have the same execution time, for the simple reason that the compiler generates exactly the same code.

```
int dotprod_size3_u50m1_unroll_propag(int v[])
{
  int rslt = 0;
  rslt += 5 * v[0];
  rslt += 0 * v[1];
  rslt += -1 * v[2];
  return rslt;
}
```

Figure 1.15. *Simple specialization of* `dotprod` *for* `size = 3` *and* `u = {5,0,-1}`

```
int dotprod_size3_u50m1_optim(int v[])
{
  return 5*v[0] - v[2];
}
```

Figure 1.16. *Optimized specialization of* `dotprod` *for* `size = 3` *and* `u = {5,0,-1}`

In fact, the same is true for the function `dotprod_size3_triv` because an optimizer such as that of gcc is capable of unrolling simple loops (option `-funroll-loops`) when the number of iterations can be determined at compile time. However, as we will see later on (see section 2.4.5), most "substantial" specializations are as yet beyond the scope of a compiler, particularly when the loops relate to complex objects.

Figures 1.15 and 1.16 show other possible specializations of the `dotprod` function relating to the partial inputs `size = 3` and `u = {5,0,-1}`. With no optimization, the functions `dotprod_size3_u50m1_unroll_propag` and `dotprod_size3_u50m1_optim` differ in terms of their execution time, but with an optimizing compilation, their equality is restored.

On the other hand, a trivial specialization such as the one shown in Figure 1.17 cannot reach the same level of performance because, while the optimizer in gcc does have the capacity to unroll loops, it can only operate on scalars and therefore cannot

dereference elements of tables even if they are known. Although in theory there is no reason why an optimizer should not be able to carry out this type of dereferencing, we are approaching the limits of what an ordinary optimizer knows how to do – limits that also serve as the starting point beyond which program specialization offers benefits in terms of performance (see section 2.4.5). We will also see that, even if its performance can be equaled by that of an optimizer, program specialization is also more advantageous in terms of predictability (see section 3.2.1).

```
int dotprod_size3_u50m1_triv(int v[])
{
  int size = 3;
  int u[] = {5, 0, -1};
  int rslt = 0;
  int i;
  for (i = 0; i < size; i++)
    rslt += u[i] * v[i];
  return rslt;
}
```

Figure 1.17. *Trivial specialization of* `dotprod` *for* `size = 3` *and* `u = {5,0,-1}`

The possible compiler of the execution platform does not only have a bearing on the execution time; it may also influence the memory space taken up. Certain data structures can be stored in the memory to a greater or lesser degree of effectiveness, and particularly of sharing (see section A.5.3.2).

1.2.4. *Opacity of the code generated*

From the point of view of specializer engineering, it is futile, during specialization, to carry out optimizations that would already be present in the compiler[5]. This would pointlessly duplicate functionalities that are complicated to fine-tune and that we may deem it better to maintain and update in a compiler.

From a practical point of view, simple optimizations are nevertheless welcome in a specializer when they allow us to improve the readability of specialized programs: `dotprod_size3` is manifestly more pleasant and easier to read than `dotprod_size3_unroll`.

However, specialized programs are not intended to be looked at. From the point of view of automatic specialization, only the generic program needs to be developed and

5 That is, assuming the exact effect of the compiler's optimizations is indeed known or can be discovered, which is not always an easy task.

maintained. A specialized program, from this point of view, holds as much (as little) interest as a binary code generated by compilation. However, in practice, it may be useful to look at a specialized program in order to fully comprehend a viewing of the specialization information (see section 3.2.2) or for debugging – which is sometimes necessary – of the specialization-optimized code (as opposed to generic code).

1.2.5. *Effect of the memory cache*

To analyze and compare the execution times of specialized programs, we can count the computations saved in relation to the original generic program. However, we should be careful not to accidentally count the computations to be carried out. Indeed, as mentioned in section A.5.1, the execution time of a program is not monotonous in relation to the computations to be carried out (instructions to be executed). In particular, if a program is too big to be contained in the memory cache (or caches), there will be cache misses, and time will be lost while the cache being executed is refereshed. A large program could therefore be far slower than a program that was more compact but capable of performing more computations.

However, as pointed out in section 1.2.2, a specialized program may indeed be significantly larger than a generic program it arises from. This can be seen in the functions in Figures 1.10, 1.11, and 1.13, even though in practice, the orders of magnitude on these particular examples are such that the cache effect will not be very noticeable, or visible at all (depending not only on the size of the cache but also on the rest of the code to be executed). On the other hand, there is a drastic increase in size with the function `dotprod_size10000` (see Figure 1.14), which is roughly 10,000 times larger than the generic function `dotprod`, whatever the value used to measure the size.

If it is able to fit in the memory cache at execution, the specialized function will be faster than the generic function. If not, it may be many times slower. By attempting to reduce the number of computations at all costs, we may occasionally cause the size of a specialized program to burgeon (known as combinatorial explosion), and hence, paradoxically, increase its execution time.

This possible increase in size is not in fact limited to loops only. It can also be found when a specializer carries out too many specializations of a subprogram for different static input values, particularly in the case of recursive inlining.

This phenomenon of dependency in the cache is not only observed for imperative programs compiled in machine language; it can also be seen, for instance, for Prolog compiled toward a virtual machine [LEU 02b].

1.3. Automatic specialization

All that has been presented in the preceding section (see section 1.1) may be implemented "by hand": given a subprogram p in a language L and a partial input $in_{\mathtt{s}}$ of p, we can seek to propagate the knowledge gleaned from $in_{\mathtt{s}}$ in p, and modify p each time we identify that a construction can be pre-executed w.r.t. L. However, this task can also be *automated*.

1.3.1. *Specializer*

The specialization of programs written in L is a transformation of programs from L to L (see section A.7.1), and a specializer is a program, written in a language L' that is not necessarily the same as L, which implements that transformation.

More specifically, a *specializer* is a program *spec* (in a language L') that, based on a program p (written in a language $L : p \rightarrow Input \rightarrow Output$) and a partial input $in_{\mathtt{s}} \in Input$ produces a specialized program $p_{in_{\mathtt{s}}}$ (written in the language L). In other words, for any program p and any partial input $in_{\mathtt{s}}$ of p, a specializer *spec* verifies the equations:

$$[\![spec]\!]_{L'}\ (p, in_{\mathtt{s}}) \quad = \quad p_{in_{\mathtt{s}}} \qquad [1.14]$$

$$[\![p_{in_{\mathtt{s}}}]\!]_{L}\ in_{\mathtt{d}} \quad = \quad [\![p]\!]_{L}\ (in_{\mathtt{s}}, in_{\mathtt{d}}) \qquad [1.15]$$

For any complete input $in = (in_{\mathtt{s}}, in_{\mathtt{d}})$ of p, we then have:

$$[\![[\![spec]\!]_{L'}\ (p, in_{\mathtt{s}})]\!]_{L}\ in_{\mathtt{d}} \quad = \quad [\![p]\!]_{L}\ (in_{\mathtt{s}}, in_{\mathtt{d}}) \quad = \quad out \qquad [1.16]$$

These equations correspond to the following diagrams, where L also appears in its de-curried form $\tilde{L}$:

$$\begin{array}{ccccc} L' : Prog' & \longrightarrow & Input' & \longrightarrow & Output' \\ \ni spec & & \supset Prog \times Input & & \supset Prog \end{array} \qquad [1.17]$$

$$\begin{array}{ccccc} \tilde{L} : Prog \times Input & & \longrightarrow & & Output \\ [\![spec]\!]_{L'} \downarrow & \searrow^{id} & & & \\ L : Prog & \longrightarrow & Input & \longrightarrow & Output \end{array} \qquad [1.18]$$

Alternatively, we can also view $in_{\mathtt{s}}$ as a parameter of the program transformation $T = [\![spec]\!]_{L'}$ and apply $T^{in_{\mathtt{s}}}$ (see section A.7.1):

$$L: \; Prog \longrightarrow (Input \longrightarrow Output) \qquad [1.19]$$

$$[\![spec]\!]_{L'}^{in_s}$$

This involves considering a curried form of $[\![spec]\!]_{L'}$ to the second argument (in *Input*), and applying it to in_s to form the program transformation $\lambda p.[\![spec]\!]_{L'}(p, in_s)$. These definitions are illustrated in Figure 1.18 where the executed programs are in rectangular boxes and the data are in ovals. Note that certain entities are treated both as data and as programs, depending on the arrow that produces or exploits them. Thus, the program p is treated as a piece of data by *spec*, which also produces the specialized program p_{in_s} as a piece of data.

For instance, a specializer for the language C must be capable, based on the source of the `dotprod` function (see Figure 1.2) and the partial static input value `size = 3`, of automatically producing the specialized function `dotprod_size3` (see Figure 1.3) or one of its variants, such as `dotprod_size3_propag` (see Figure 1.11).

A program p_{in_s} specialized in this way is sometimes also called a residual program and is denoted as p_{res}. This nomenclature refers to the fact that the specialized program is generally the result of a transformation process during which fragments of the program are precomputed (replaced by the result of their evaluation), and the residual program is what is left when there is nothing more to pre-execute.

Specializers are also sometimes denoted by *mix* rather than by *spec*, which comes from the notion of *mixed computation* [ERS 78]. The principles for building a specializer using program transformations are provided in Chapter 2.

(A variant of this definition of a specializer is given in section 3.4.3, where an argument in_{sd} of the program may be considered to be static *and* dynamic, and serve both for creating the specialized code $p_{in_s,in_{sd}}$ and for executing it $[\![p_{in_s,in_{sd}}]\!]\ (in_{\mathsf{sd}}, in_{\mathsf{d}})$. This variant enables us to speak of cases where a piece of information cannot easily be integrated into a program's code).

NOTE 1.2.– The form of specialization we have just defined corresponds to *automatic specialization* in the sense that it is enough to provide a generic program and static inputs. However, it is also possible to *program* specialization, i.e. to explicitly program the generation of the specialized code.

By comparison, owing to the programming effort needed, we rather consider this approach as *semi-automatic specialization*. This other form of specialization also shows a number of disadvantages in relation to automatic specialization (see section 2.4.6).

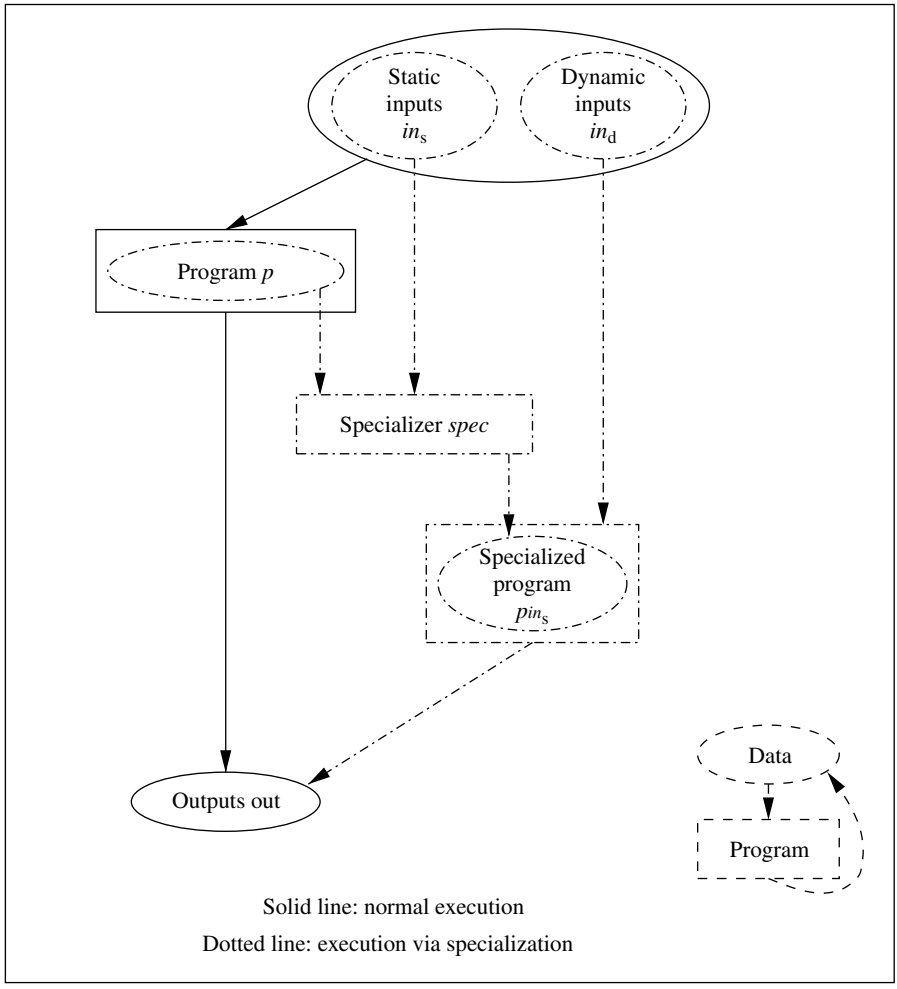

Figure 1.18. *First Futamura projection*

1.3.2. *Operation of specialization*

A specializer *spec* for the language L is a program in the language L' like any other; it is executed in a certain model of execution M' for L': $exec_{M'}\ spec\ (p, in_s)$. In keeping with our conventions, when the execution model is not ambiguous in the context, the model indicator is omitted: $exec\ spec\ (p, in_s)$.

The *operation of specialization* splits an execution into two successive parts. The first part is the creation of the specialized program; this is carried out according to an execution model M' of L', on a *specialization platform*, i.e. on the execution

platform of a specializer: $exec_{M'}\ spec\ (p, in_{\mathsf{s}})$. The second part, $exec_{M}\ p_{in_{\mathsf{s}}}\ (in_{\mathsf{d}})$, is the execution of the specialized program; this is carried out according to an execution model M of L, on an *execution platform* that is not necessarily the same as the specialization platform but that is generally the platform on which the generic program would have been executed had it not been specialized beforehand.

When the specialization platform and execution platform are the same (which implies that $L = L'$), transformation of *spec* programs can use the language's own execution mechanisms to carry out the precomputations in p on the data in_{s} (see section 2.1.5). When the specialization platform and execution platform for the specialized program are different, e.g. when $L \neq L'$, we can speak of *cross-specialization*, in a similar sense to cross-compilation. In this case, in order to carry out the precomputations in p on the data in_{s}, there has to be a means in L' of executing code fragments written in L, e.g. an L-interpreter in L' (see section 2.1.5).

1.3.3. *Execution times*

We said that specialization split the execution of a program (or subprogram) into two successive parts: execution of the specializer and execution of the specialized program. These two executions have two corresponding *execution times*: *specialization time* and *(specialized) run time*. Note: Specialization time as defined here must not be confused with the *moment* of specialization (at compile time or run time, see section 1.5).

The term *static* is generally applied to what is done before the actual execution of the (specialized) program, and *dynamic* to what is done once the program has launched. For instance, we can draw a parallel between static and dynamic memory allocations (see section A.1.2), also carried out, respectively, before the commencement of execution and during the execution.

In particular, we use the term *static computation* to denote any precomputation carried out during specialization (see section 1.1.4.2), and *dynamic computation* to denote any computation carried out by the specialized program. We term the code in the generic program corresponding to static computations the *static slice*, and code fragments in the generic program corresponding to dynamic computations the *dynamic slice*. These notions acquire a stronger meaning in the context of offline specialization (see sections 3.1.2 and 3.1.6).

1.3.4. *Advantages and disadvantages to automatic specialization*

Manual specialization (writing a specialized program by sight, based on a generic program) is a long, tedious, and complex task, even for small programs

(see section 5.1). It requires expertise both in the field of application of the programs and in specialization methodologies. Owing to this complexity, it is easy to make mistakes when trying to identify precomputable terms and carrying out the precomputations; in this case, the specialized program created does not conserve the semantics of the original program exactly. In addition, this type of error may remain undiscovered for a long time. It is also possible to "miss" (not to see) *specialization opportunities*, and thus produce underspecialized programs. Finally, it is a task that must be repeated each time the generic program or static input changes, which is a more or less frequent operation depending on the lifecycle of the software. Hence, the code is more difficult to maintain because there are numerous variants that exist simultaneously (generic code and specialized versions).

On the other hand, *automatic specialization* reduces the need for human intervention, and for expertise: an expert *user of a specializer* can prepare an automatic specialization once and for all so that a non-expert *user of the specialization* can effortlessly produce specialized versions at will for different values of the static inputs, or even make small modifications to the generic program afterward. The user of the specialized program is usually also a *user of the specialized code*.

Automatic specialization also ensures[6] that the specialized code behaves in exactly the same way as the generic code. In particular, the level of security is the same as that of the original program. Automatic specialization also guarantees systematic specialization, at least within the limits of the capability of the specializer in question (see Chapter 6); there is no risk of overlooking code fragments to be specialized. With certain techniques, the degree of specialization can even be predicted or controlled before any actual specialization is carried out (see section 3.2.1). Finally, the issues of maintenance are also solved because only the generic code needs to be maintained and adapted; specialized variants are obtained automatically.

There are only a few exceptional cases (only very incompletely studied, if at all) where the level of security or reliability can be altered, if care has not been taken to prevent specialization deleting certain *deliberate*[7], apparently superfluous computations or redundancies. This is not peculiar to specialization; it is a general characteristic of optimizing program transformations. For instance, the issue is just as prevalent for compilers, which may factor repeated computations by common subexpression elimination.

---

6 That is, of course, assuming the specializer is correct, in the same way that a compiler or an interpreter is always assumed to be so.

7 Superfluous computations allow us to "noise" the execution of sensitive systems (such as chip cards) in order to restrict the deductions that can be made on the program or its data by monitoring electromagnetic radiation, electrical consumption, etc. As for redundant computations, they protect against harware or software errors, and against deliberate interference, e.g. in terms of the power supply, which favors the observation of information that is supposed to remain hidden during normal execution of the program.

However, the degree of automation needs to be relativized. To be implemented and exploited, the functionality of specialization must in practice be accompanied by various peripheral tasks (see section 12.2), which are still partly manual (see section 12.3) and which do require a certain degree of expertise. In particular, it is still largely intrusive: usually a program has to be manually modified in order to implant specialization within it, making the process of software engineering more arduous. Also, the fact that automatic specialization is systematic may also be a disadvantage, particularly with excessive loop unrolling (see section 1.2.5). A similar problem arises if the same subprogram is called with a large number of different partial static inputs: an excessive number of specialized subprograms (which is problematic or not financially viable) may then be produced by recursive inlining. However, techniques to avoid this problem are presented in Chapter 11.

1.4. Main applications of specialization

Program specialization is a "transverse" optimization technique, independent of any particular field of application. In Chapter 5, we will look at a number of examples in different fields: exploitation and network systems, scientific calculations, graphics, and software engineering. We might also cite lexical and syntactical analysis (parsing) (see below), ecological simulation [AND 94], simulation of memory caches and microprocessors [GRA 99, GRA 00a], Web services [MAO 04a] or grid services [MAO 04b], etc. However, there is one field of application for which specialization is particularly apt: compilation using an interpreter.

1.4.1. *Application 1: compiling using an interpreter*

A specializer enables us to automatically create a compiled program using a simple interpreter.

Let *interp* be a source language L_{src} interpreter written in an object language L_{obj}. By definition (see section A.4.2.1), it verifies $[\![\mathit{interp}]\!]_{L_{\mathsf{obj}}}\ (p_{\mathsf{src}}, in_{\mathsf{src}}) = [\![p_{\mathsf{src}}]\!]_{L_{\mathsf{src}}}\ in_{\mathsf{src}}$ for any program p_{src} and input in_{src} of L_{src}. If we have a specializer *spec* for the language L_{obj}, written in a language L, then we can specialize the program $interp$ in L_{obj} to a partial input that is a program p_{src} in L_{src}: thus, we get $\mathit{interp}_{p_{\mathsf{src}}} = [\![\mathit{spec}]\!]_L\ (\mathit{interp}, p_{\mathsf{src}})$. The resulting specializer interpreter then verifies $[\![\mathit{interp}_{p_{\mathsf{src}}}]\!]_{L_{\mathsf{obj}}}\ in_{\mathsf{src}} = [\![p_{\mathsf{src}}]\!]_{L_{\mathsf{src}}}\ in_{\mathsf{src}}$. We can therefore recognize in $\mathit{interp}_{p_{\mathsf{src}}}$ the result of a compilation of the source program p_{src} in the object language L_{obj} (see section A.4.3.1). In other words:

$$p_{\mathsf{obj}} \quad = \quad [\![\mathit{spec}]\!]\ (\mathit{interp}, p_{\mathsf{src}}) \qquad [1.20]$$

This equation, illustrated in Figure 1.19, is called the *first Futamura projection* [FUT 71]. The compiled program p_{obj} indeed verifies $[\![p_{\mathsf{obj}}]\!]_{L_{\mathsf{obj}}} = [\![p_{\mathsf{src}}]\!]_{L_{\mathsf{src}}}$.

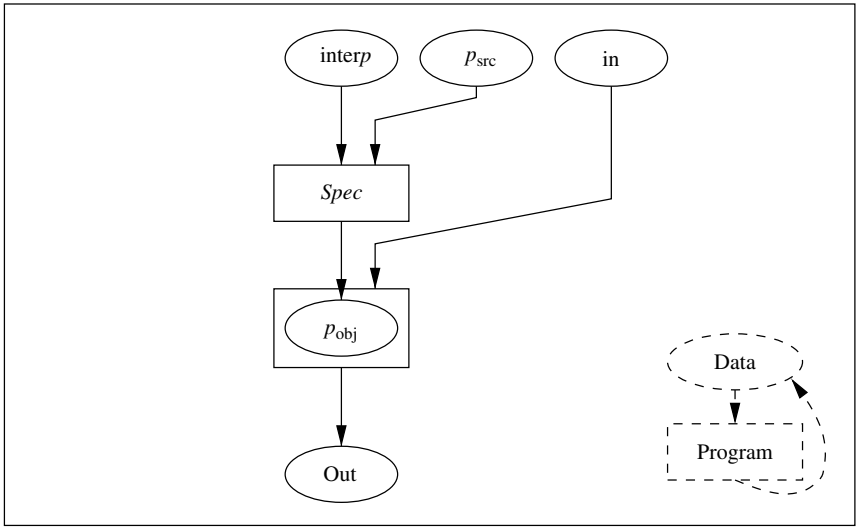

Figure 1.19. *First Futamura projection*

In practice, although compiled programs p_{obj} formed in this manner cannot always compete with compiled programs produced by L_{obj}-dedicated optimizing compilers, they often perform well – better, at any rate, than would the interpretation of p_{src} [THI 00]. This is even more so when the basic instructions of the language L_{obj} perform operations that are so different from one another that there is no universal optimizing rewriting scheme, as is often the case when the object language commands are coarse-grained. See section 5.3 for concrete results regarding the specialization of interpreters.

Consider the case where *interp*, a program written in L_{obj}, is a layered L_{src}-interpreter in terms of an object language L'_{obj}, a sublanguage of L_{obj} (see section A.4.2.4). Hence, if we prevent the pre-execution of the building blocks of L'_{obj} (see section 8.3.5), the specialization $p_{\mathsf{obj}} = [\![spec]\!]_L\ (interp, p_{\mathsf{src}})$ may enable us to compile from L_{src} into the sublanguage L'_{obj}, which is trickier than simply compiling into L_{obj}. This is the case when all the computations written in L_{obj} that relate to the running of a fragment of the program p_{src} and its (JIT) translation in terms of building blocks of L'_{obj} can be pre-executed. The execution of these building blocks, which in any case may also relate to the input in_{src}, remains differed hypothetically. If the specializer also manages to eliminate everything relating to the running and JIT translation of p_{src}, what remains in p_{obj} is made up purely of unexecuted building blocks of L'_{obj} and, therefore, belongs to L'_{obj} rather than L_{obj}: the interpretation layer is thus eliminated.

```
void mini_printf(char fmt[], int val[])
{
  int i = 0;
  while (*fmt != '\0') {
    if (*fmt != '%')
      putchar(*fmt);
    else
      switch (*++fmt) {
        case 'd': putint(val[i++]); break;
        case '%': putchar('%');     break;
        default : prterror(*fmt);   return;
      }
    fmt++;
  }
}                                          // Bold: pre-executable fragments
```

Figure 1.20. *Pre-executable fragments of* `mini_printf` *when* `fmt` *is known*

To illustrate this, let us look once more at the function `mini_printf`, the display format interpreter presented in section A.4.2.4. The source language L_{src} here is the display format language, the object language L_{obj} is C, and the (sub-) display language L'_{obj} is built on the foundations formed by the display functions `putint`, `putchar`, and `prterror`, as well as the table-reading operation `val[]`. The commands in `mini_printf` that can be pre-executed when a format `fmt` is known are shown in bold in Figure 1.20. The specialization of `mini_printf` to a format `fmt = "<%d,%d>"` is shown in Figure 1.21. It corresponds to the compilation of the display format `"<%d,%d>"` in terms of the display language L'_{obj} (see section A.4.3.1). In reality, the specialization compiles a display format in a program of the abstract display machine.

Other cases of interpretation of "mini-languages" in that sense may benefit from compilation by specialization. For instance, a regular-expression interpreter may be compiled by specialization in a program that more or less represents an automation [BON 90, MOG 92]. Specific LR parsers can also be generated from a generic parser [SPE 00].

Similarly, a naïve implementation of a "character-string interpreter" (which searches for occurrences of a character string in a text), of quadratic complexity (length of the string searched for times length of the text), may in a manner of speaking be compiled by specialization to automatically produce a program that implements an effective searching algorithm, of linear complexity (in the length of the text), such as the Knuth-Morris-Pratt (KMP) algorithm [AGE 02, AGE 06, CON 89, KNU 77] or the Boyer-Moore algorithm [BOY 77, DAN 06b].

```
void mini_printf_fmt(int val[])
{
  putchar('<');
  putint(val[0]);
  putchar(',');
  putint(val[1]);
  putchar('>');
}
```

Figure 1.21. *Specialization of* `mini_printf` *for* `fmt = "<%d,%d>"`

Not every specializer, in the general sense of the term, is necessarily capable of eliminating interpretation in a program, i.e. the running of static data (see section A.4.2). For instance, this is not possible with a *trivial specializer*, i.e. a specializer that produces only trivial specialized programs. A specializer that is capable of getting rid of interpretation is said to be *Jones-optimal* [MOG 00].

1.4.2. *Application 2: transforming an interpreter into a compiler*

A specializer also enables us to automatically create a compiler based on an interpreter.

Suppose, for this purpose, that the object language L_{obj} (in which the interpreter *interp* is written) and the language L (in which the specializer *spec* is written) are identical.

We can then define a program *comp* in $L = L_{\text{obj}}$ as follows:

$$comp \quad = \quad [\![spec]\!]\ (spec, interp) \qquad [1.21]$$

This other equation is called the second Futamura projection. It describes how *spec* enables us to automatically create a compiler *comp* from L_{src} to L_{obj}, written in L_{obj}, if we have an L_{src} interpreter *interp*, also written in L_{obj}. The program *comp* is defined as the specialization of the specializer itself, to the interpreter. Hence, we have $comp = spec_{interp}$ and $[\![comp]\!]\ p_{\text{src}} = [\![spec_{interp}]\!]\ p_{\text{src}} = [\![spec]\!]\ (interp, p_{\text{src}}) = p_{\text{obj}}$. The program *comp* created in this way is therefore indeed a compiler (see section A.4.3.1):

$$[\![comp]\!]_{L_{\text{obj}}}\ p_{\text{src}} \quad = \quad p_{\text{obj}} \qquad [1.22]$$

$$[\![p_{\text{obj}}]\!]_{L_{\text{obj}}}\ in_{\text{src}} \quad = \quad [\![p_{\text{src}}]\!]_{L_{\text{src}}}\ in_{\text{src}} \qquad [1.23]$$

These equations are illustrated in Figure 1.22. The possibility of applying a specializer to itself is called *auto-application*.

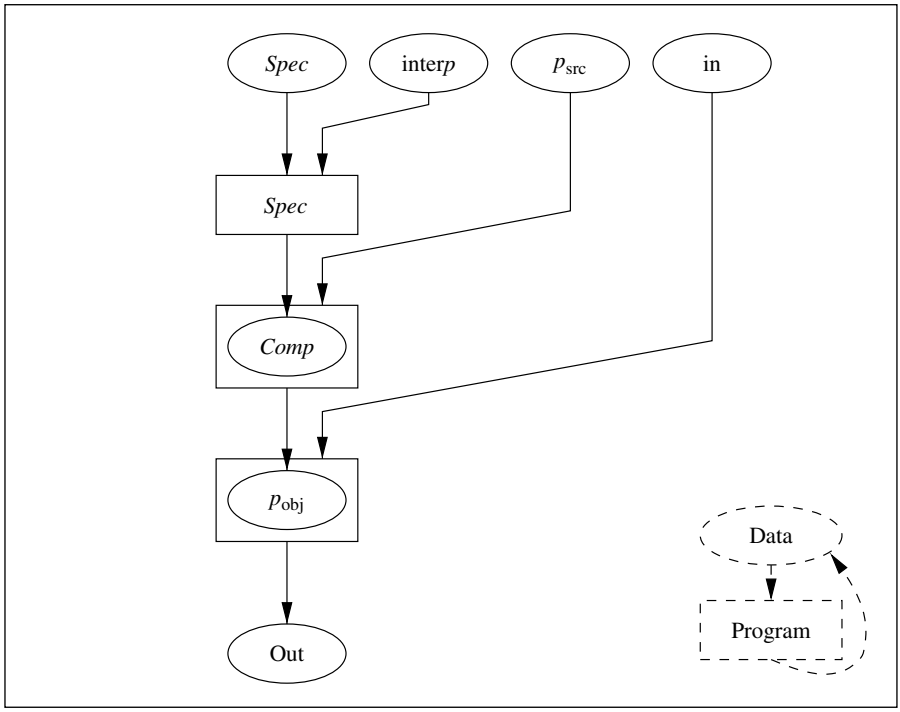

Figure 1.22. *Second Futamura projection*

Note that such a compiler *comp* generates exactly the same compiled programs as those in the first Futamura projection. Hence, the issues relating to the code's performance are identical.

The important point arises from the fact that it is always easier to write a code that carries out actions (e.g. an interpreter) than a code that generates a code that carries out actions (e.g. a compiler). The advantage of the second Futamura projection is that it enables us to develop the easy-to-write code of an interpreter and automatically produce the difficult-to-write code of a compiler.

1.4.3. *Application 3: creating a compiler generator*

Finally, also based on an interpreter, a specializer enables us to automatically create a *compiler generator* (or *compiler*).

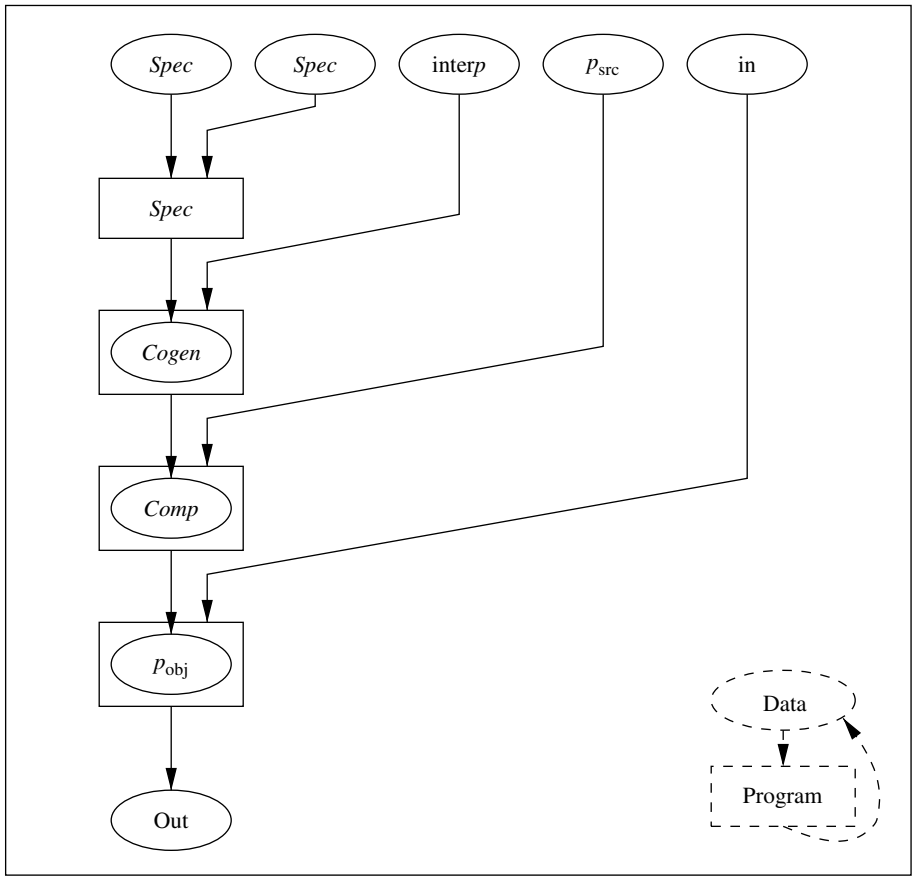

Figure 1.23. *Third Futamura projection*

To illustrate this, let us again suppose that the object language L_{obj} (in which the interpreter *interp* is written) and the language L (in which the specializer *spec* is written) are identical. We can then define a program *cogen* in $L = L_{\mathsf{obj}}$ as follows:

$$cogen \quad = \quad [\![spec]\!]\,(spec, spec) \qquad [1.24]$$

This final (or perhaps not [GLÜ 09]) equation is the *third Futamura projection*. It expresses the way in which *spec* enables us to construct a program *cogen* that automatically generates L_{src} compilers in L_{obj}. This compiler generator is defined as the specialization of the specializer to itself; therefore, we have $cogen = spec_{spec}$. The program *cogen* formed in this manner generates compilers based on interpreters:

$$[\![cogen]\!]\ interp \quad = \quad comp \qquad [1.25]$$

Indeed, we have $[\![cogen]\!]\ interp = [\![spec_{spec}]\!]\ interp = [\![spec]\!]\ (spec, interp) = comp$. These equations are illustrated in Figure 1.23.

As above, such a compiler generator can only generate compilers that correspond to the second Futamura projection, which, in turn, generate the same compiled programs as those in the first Futamura projection. The remark made above about the performance of the compiled programs produced therefore also applies here (also see Glück's works [GLÜ 91, GLÜ 94b, GLÜ 09] for a general overview of the Futamura projections). Depending on the techniques used, it may sometimes be beneficial to write *cogen* directly rather than writing *spec*, in order to eliminate redundancies that decrease the specializer's performance (though not the performance of the specialized programs computed) [BIR 94, THI 96].

Also see section 3.1.7 for the definition, in the context of offline specialization, of a compiler generator also called *cogen*, but which differs from what has been defined here.

1.5. Specialization times

We can distinguish two particular specialization times, which are two key moments when specializations can be created during the course of the development and execution of a program: "compile time" and "run time"[8].

These two specialization times are not mutually exclusive; they can be implemented simultaneously within the same program (see, e.g. section 9.3.2).

NOTE 1.3.– This "specialization time" must not be confused with the "specialization time" (moment of specialization – see section 1.3.3), which for its part denotes the moment when the specialized code is constructed (specialization time) when we want to speak of it in comparison to the moment when the specialized code is executed (run time).

1.5.1. *Compile-time specialization*

Compile-time specialization is a specialization carried out some time before the program is executed.

8 We could also define other specialization times: *load-time specialization* and *link-time specialization*; link-time specialization may be "static" (before execution) or "dynamic" (during execution, see section A.4.1). The techniques to be used in order to exploit these other specialization times largely resemble those for compile-time and runtime specialization. However, some are specific, as is the resolution of symbols (introduction of the address values) at link time. Also see Chambers' works as regards staged compilation [CHA 02].

The advantage of this type of specialization is that we can take an arbitrary amount of time (and more generally, an arbitrary amount of resources) to create an efficient specialized program. The process of generation itself does not have to be particularly optimized, and we can afford to seek out the best optimizations [BOD 98, PAR 02a]. However, the only static input we can exploit in order to specialize a (sub)program in this way is data that must be known far enough in advance of execution. However, *optimistic specialization* circumvents this limitation by prespecializing the program (at compile time) to the most common static inputs (see section 9.3).

Compile-time specialization is generally a transformation from a source code (of a generic subprogram) to a source code (of a specialized subprogram). By incorrect use of language, the term "compile-time specialization" has come to denote any source-to-source specialization. Nevertheless, generating specialized source code, immediately compiled and then integrated into a running program, has also been put forward as a runtime execution technique [BHA 04b] (see section 1.5.3).

1.5.2. *Runtime specialization*

Runtime specialization, or *JIT specialization*, is a specialization that is carried out during the actual execution of a program.

The advantage of runtime specialization is that it enables us to exploit information that is unknown until after the program is launched, e.g. data received over a network or interactively entered by the user. The disadvantage of this type of specialization is that the time required to create a specialized subprogram must be taken into account in the total run time of the program. Consequently, the cost of producing this specialized code, which is more efficient than the generic code, is only covered if it is executed enough times (see section 1.6).

Runtime specialization is generally a transformation into immediately executable machine code, e.g. by an indirect function call in the case of C. Directly generating binary code avoids a costly additional call to a JIT compiler, or to a standalone compiler. By incorrect use of language, the term "runtime specialization" has come to denote any direct specialization into machine code without going through source code, once the program has been launched. However, as indicated above, certain forms of runtime specialization do go through the source code [BHA 04b]; others do away with the analysis of the source code by deforestation but retain most of the phases of a compiler before producing machine code [SPE 97b]. Still others operate on an intermediary manifestation [MAS 02].

Currently, in general, the performance of compilation platforms is such that generating the source code and compiling it JIT with a static compiler becomes an acceptable alternative in certain particular contexts, such as for dynamic programming

languages. Hence, some platforms offer dynamic compilation of script languages by generating C source code, and also because this provides, "for free", a degree of portability that is otherwise extremely expensive for direct generation of binary code for different environments and material architectures [WIL 09].

A form of runtime specialization that produces machine code directly is presented in section 10.4.1. In this chapter, we only illustrate its use in the case of C. In terms of the interface, this runtime specialization produces dedicated specializers, i.e. specializers that are specific to a particular configuration of static and dynamic data.

Thus, for example, we might have a function `dotprod_size_gen` that generates runtime specializations of `dotprod` to the static parameter `size`. In concrete terms, `dotprod_size_gen` returns a pointer to a function with two arguments (the dynamic parameters `u` and `v` of `dotprod`) that, in turn, returns an integer. In other words, the function `dotprod_size_gen` has the following signature:

```
typedef int ((*ii2i)(int,int));
ii2i dotprod_size_gen(int size);
```

Thus, it is used:

```
// The value of the size is arbitrary; it can be the result of a computation
sizex = exp;
...
// Generation of the specialized code
dotprod_size = dotprod_size_gen(sizex);
...
// Use of the specialized code
r = (*dotprod_sizex)(ux, vx);
```

More complex forms of exploitation are detailed in Chapter 9.

1.5.3. *Specialization server*

Specializing a code fragment is a very clearly delimited task, which can be carried out in parallel with the rest of the execution of a program: in a separate thread or process, or on another processor or even another machine. In this case, the financial viability of specialization can be improved in two ways: by reducing the time taken for specialization (under certain conditions) and enhancing the quality of optimization of the specialized code produced.

This idea can be put into practice in the form of a *specialization server*, accessible over the network, which produces specialized code on demand following a specialization request [BHA 04b]. This type of *delocalized specialization* is particularly pertinent in the case of onboard systems where resources (time, space,

and electricity) are limited [BHA 04b]. Specializations that would be impossible to carry out directly on the onboard system because of the lack of resources then become possible via the server. Even those specializations that could be carried out on the onboard system itself can be performed more efficiently on the specialization server (depending on the server's power, its load, the size of the data to be exchanged, the number of precomputations to be carried out, the bandwidth and the latency of the network, etc.).

The specialized codes produced may also be optimized further. Indeed, as users, we are often prepared to accept a little delay for a function to start, in the interests of better quality of service once that function is properly initiated (e.g. to watch a movie). However, for such a form of specialization to become financially viable, the specialized code has to be used a great many times (see section 1.6). Also, in the case where we can make requests to several specialization servers, we are dealing with *distributed specialization* rather than delocalized specialization [SPE 97a].

When the value of the static data is discovered a certain number of times before the subprogram that uses them is actually invoked, it is useful to make calls for *anticipated specialization* in the hope that the specialized code will already be available when the subprogram is called. If the specialized code is not yet available at that point, we can, depending on the strategies, wait for it to become available or use a default subprogram – e.g. the (non-specialized) generic subprogram. On condition that adequate clustering is available, the specialized codes produced can also be conserved and reused, or shared with other running programs.

However, delocalized specialization involves dealing with additional problems: cross-specialization because the specialization server and the machine or program that is run may have different processors and execution environments; data-sharing with the execution in progress, which may necessitate additional exchanges over the network (with or without data caches); and static calls to system functions, which must be made, at the server's request, on the client execution machine that initiated the specialization request.

In terms of the order of magnitude, the time taken to carry out a delocalized specialization may be approximately one second, with two-thirds of this time being spent generating the code, and the remaining one-third for the transfers over the network [BHA 06, BHA 08].

1.5.4. *Specialized code cache*

Cache techniques, similar to the technique of memorization [MIC 68], can be used when we have to specialize the same function several times over, be it on a specialization server (see section 1.5.3) or in the more usual case of a simple "locally"

executed specialization. In return for a (usually only slightly) increased expense to look for already-generated specialized codes, when the contexts of specialization are identical, we avoid costly and pointless code productions.

In this *specialized code cache* (or *specialization cache*), runtime specializations coexist with compile-time specializations carried out optimistically to cover frequent cases (see section 9.3). These frequent cases may be known if we have prior knowledge of the domain and the code, or can be empirically discovered using profiling.

Like any cache, a specialization cache has to handle not only additions but also deletions (of specialized code). For this purpose, besides standard *cache strategies*, strategies specific to runtime specialization have been studied with a view to limiting the cache size [VOL 97]. Cache strategies linked to the relative invariance and the way in which the specialized code is used (see Chapter 9) have also been put forward [GRA 00b]. Techniques for identifying the contexts of specialization based on chopper functions have also been used in the context of a specialization server [BHA 06, BHA 08]. A specialized code cache server (memorization master) has even been proposed in the context of distributed specialization [SPE 97a].

The notion of a specialized code cache may also be compared to that used in online specialization, where it is an integral part of the code generation process; it enables us to ensure the termination of a sequence of reductions by creating recursive call code [RUF 91, WEI 91]. The same kind of mechanism also enables us to terminate specializations and share specialized functions in the context of offline specialization (see Chapter 3).

1.6. Financial viability of specialization

Two main parameters need to be taken into account in order to evaluate the *profitability of automatic program specialization*: the quality of the resulting specialized program (how it is better than the original generic program) and the resources consumed in order to transform the generic program into a specialized program – particularly the specialization time and the memory space taken up.

In addition to a few definitions, we also give some *orders of magnitude* of practical results obtained using Tempo, a specializer for C that is presented in Chapter 4. It would be imprudent to draw too hasty conclusions from these, given that the results can vary depending on the type of program, the choice of static inputs, the execution platform, etc. In addition, the gains presented here are only measured on the code fragment being specialized, rather than on the entire program into which the specialized code is inserted. Nevertheless, these orders of magnitude enable us to "clarify our ideas". More concrete figures are given in Chapter 5, which recounts some genuine experiences with Tempo.

1.6.1. *Specialization gain*

The *specialization gain* is a comparison of a measure of resource consumption between a generic program p and a specialized program $p_{in_s tat}$. Unless explicitly stated, this comparison is generally expressed as the ratio between the measurement for the generic program and the measurement for the specialized program. The gain is greater than one if the performance is improved, and less than one if it is hindered.

1.6.1.1. *Gain in terms of time*

Unless otherwise indicated, the resource measured is generally the execution time, on a given execution platform M and for given sets of input values $(in_{\mathtt{s}}, in_{\mathtt{d}})$. This *time gain* therefore expresses an increase in the speed of execution, i.e. a reduction in execution time, also called *speedup*:

$$\text{time gain} \quad = \quad \frac{\mathit{time}[\mathit{exec}_M\ p\ (in_{\mathtt{s}}, in_{\mathtt{d}})]}{\mathit{time}[\mathit{exec}_M\ p_{in_{\mathtt{s}}}\ (in_{\mathtt{d}})]} \qquad [1.26]$$

The "gain" may be less than one if the performance is adversely affected, e.g. in case of cache misses if a loop is unrolled too much, or a recursion is excessively inlined (see section 1.2.5).

To give some idea of the orders of magnitude, with the specializer Tempo, the gain in execution speed for compile-time specialization varies, in practice, from a few percent to a factor of 10, with points at nearly 100 (generally for very small functions). The time gain for runtime specialization, for its part, typically varies from a few percent to a factor of five, with points at 40. Because of the techniques used to generate code rapidly, a runtime specialized program is generally slower than one specialized (and highly optimized) at compile time, by a factor that varies between 1.1 and 1.5, with points at 4.

1.6.1.2. *Gain in terms of space*

The *gain in terms of space* (program size, see section A.5.2) may be formulated in a number of different ways, depending whether we are interested in the static memory:

$$\text{space gain (1)} \quad = \quad \frac{\mathit{size}[p]_M}{\mathit{size}[p_{in_{\mathtt{s}}}]_M} \qquad [1.27]$$

the dynamic memory:

$$\text{space gain (2)} \quad = \quad \frac{\mathit{size}[\mathit{exec}_M\ p\ (in_{\mathtt{s}}, in_{\mathtt{d}})]}{\mathit{size}[\mathit{exec}_M\ p_{in_{\mathtt{s}}}\ (in_{\mathtt{d}})]} \qquad [1.28]$$

or indeed both of them:

$$\text{space gain (3)} \quad = \quad \frac{size[p]_M + size[exec_M\ p\ (in_s, in_d)]}{size[p_{in_s}]_M + size[exec_M\ p_{in_s}\ (in_d)]} \qquad [1.29]$$

In practice, it is usually the gain in terms of size of the static memory that is reported. Also in practice, in that it is more commonplace to specialize subprograms than entire programs, this size must include that of the extra code written into the program in order to exploit the specialized code (see Chapter 9).

To give the reader some idea of the orders of magnitude, with Tempo, the gain in terms of the size of the code (for the fragment being specialized) can in certain cases attain a factor of 10, equating to a space gain of up to 30% for the whole program.

1.6.2. *Specialization time*

The *specialization time* (or *specialization rate*) is the execution time $time[exec\ spec\ (p, in_s)]$ of a specializer *spec* for a program p and a static input in_s to produce a program p_{in_s}. The specialization time is often a not-very-important parameter in compile-time specialization because it is generally negligible in comparison to all the other tasks of development. Similarly, compilation time does not tend to be taken into account except for very large systems, and in these cases more attention is paid to the incrementality of separated compilation than to the individual compilation times (However, see section 1.6.4 for an example where the program to be specialized must be executed as soon as possible once the input values are known).

However, the specialization time is a crucially important parameter for runtime specialization because the time taken to generate the specialized code is included in the total execution time of the program (except with a separate specialization server, see section 1.5.3).

It must also be borne in mind that, depending on the program and the static inputs, the amount of precomputations carried out during a specialization operation, and hence the specialization time, may be arbitrarily large. For example, depending on how a program for searching for character strings within a text (see section 1.4.1) is written, the specialization computations may be linear or quadratic along the length of the string [AGE 06]. In this manner, the specialization time differs from the compilation time, which is usually fairly limited (even if it is not necessarily linear), and which depends only on the program and not on the value of certain inputs.

In terms of orders of magnitude, with Tempo, compile-time specialization is rarely faster than a tenth of a second. Indeed, *a priori*, the source code of the generic program must be read, analyzed and transformed, and the resulting specialized

program must be written. Even though a fair number of operations can be carried out in advance and factored (particularly the analysis of the program in the context of offline specialization, see Chapter 3), in practice, this entails reading from and writing to disk, creating files, manipulating symbolic data (character strings and data structures that represent programs), and translating formats. Depending on the context (see sections 1.5.3 and 2.1.5), a compile-time specialization may also include compiling and editing links.

On the other hand, using techniques based on assembling precompiled code fragments, the time consumed by runtime specialization may drop to less than a microsecond for very simple programs [MAR 99a] because it is possible to reduce the operations needed to a few copies and memory-writes. The specialization time may be less than a millisecond for a variety of algorithms used in scientific computations and in image processing [NOË 98]. The time taken for memory allocations may have to be added to this figure, depending on the management of the memory space that is regained by creating and storing specialized programs (see section 1.5.4).

1.6.3. *Size of the specializer*

As regards compile-time specialization, the *size of the specializer* is usually unimportant because the specialization generally takes place on platforms with abundant memory, even if the specialized program generated must then be executed on a system where the memory is limited (which may, however, cause problems of cross-specialization, see section 1.3.2). Similarly, we rarely concern ourselves with the size of a compiler.

On the other hand, for runtime specialization and if we are using a system with limited memory, we must take account of the size of the dedicated specializer or specializers (see section 3.1.6) that are incorporated into the program so they can be called. Account must also be taken of the size of the extra code written into the program to exploit the specialized code, starting with the call to the specializer and the calls to the specialized codes generated (see Chapter 9). The possible specialized code cache system must also be taken into account.

In addition, similarly as for all program sizes, the sizes above may be broken down into static sizes and dynamic sizes.

1.6.4. *Specialization before execution*

If we have to execute a program p on a known input *in* and wish to minimize the execution time, it may sometimes be advantageous to begin by specializing it to

a partial input in_{s} of $in = (in_{\mathsf{s}}, in_{\mathsf{d}})$. This operation is profitable if the following condition is satisfied:

$$\begin{aligned} \mathit{time}[\mathit{exec\ spec}\ (p, in_{\mathsf{s}})] + \mathit{time}[\mathit{exec}\ p_{in_{\mathsf{s}}}\ (in_{\mathsf{d}})] \\ < \quad \mathit{time}[\mathit{exec}\ p\ (in_{\mathsf{s}}, in_{\mathsf{d}})] \end{aligned} \qquad [1.30]$$

This condition may be fulfilled, for example, when the input in_{s} is a piece of information that is constantly being used in p's computations: in this case, a great deal of time is saved by eliminating all computations involving in_{s} from p. Hence, it is sometimes profitable to specialize a program before executing it, even if it is only executed once on that data set. This observation chiefly relates to compile-time specialization, but it is also applicable to runtime specialization of a function of a program.

Repeat interpretation is common in this scenario (see section A.4.2.4). If a program p_{src} in a language L_{src} performs a great many computations, e.g. if it contains loops or recursions, a lot of time will be spent in an interpreter *interp* of L_{src} to repeatedly decode the same instructions. In this case, instead of proceeding to execute the interpreter directly, we can first specialize that interpreter to the program, then execute the resulting specialized interpreter *interp* p_{src}:

$$\mathit{exec\ interp}(p_{\mathsf{src}}, in_{\mathsf{src}}) \Longrightarrow \begin{cases} \mathit{interp}_{p_{\mathsf{src}}} := \mathit{exec\ spec}\ (\mathit{interp}, p_{\mathsf{src}})\,; \\ \mathit{exec\ interp}_{p_{\mathsf{src}}}(in_{\mathsf{src}}) \end{cases} \qquad [1.31]$$

This is exactly what we do when we wish to compile a program before executing it, rather than interpreting it. At any rate, the specialized interpreter $\mathit{interp}_{p_{\mathsf{src}}}$ has the status of compiled code p_{obj} as per the first Futamura projection (see section 1.4.1).

However, for programs that perform few computations, e.g. in "languages" that do not have iterative constructs, such as the display format language discussed in section A.4.2, specialization takes too long in comparison to the actions of the interpreter. In this case, if the program is only executed once, it is not financially viable to specialize it before executing it; it is better to interpret directly.

However, the specialization time can be reduced by using a strategy of offline specialization (see Chapter 3), particularly with interpretation of specialization actions (see section 3.1.5) or creation of a generating extension (see sections 3.1.6 and 3.1.7). In the particular case of an interpreter, we can also speed specialization up by using a dedicated compiler obtained through auto-application (see section 1.4.2, second Futamura projection), for an identical specialized code $\mathit{interp}_{p_{\mathsf{src}}}$ produced.

1.6.5. *Runtime specialization and break-even point*

More generally, the profitability of specialization depends on the number of uses of the specialized program: the cost of creating it may be recouped over several executions. Thus, if we have to successively execute a (sub)program p on each of the inputs $in_i = (in_{\mathtt{s}}, in_{\mathtt{d},i})$ for $1 \leqslant i \leqslant n$, it is profitable to first specialize p to the partial input $in_{\mathtt{s}}$ on the following condition:

$$\textit{time}[\textit{exec spec } (p, in_{\mathtt{s}})] + \sum_{i=1}^{n} \textit{time}[\textit{exec } p_{in_{\mathtt{s}}}\ in_{\mathtt{d},i}] < \sum_{i=1}^{n} \textit{time}[\textit{exec } p\ (in_{\mathtt{s}}, in_{\mathtt{d},i})] \quad [1.32]$$

Assuming the specialized program is faster than the original program, there is still a minimum number of uses of the specialized program beyond which specialization becomes profitable; this is called the *profitability threshold* or *break-even point*. Thus, to simplify, when the execution time does not depend on the actual value of the different partial inputs $in_{\mathtt{d},i}$, specialization is profitable beyond the following number of executions:

$$\text{break-even point} = \frac{\textit{time}[\textit{exec spec } (p, in_{\mathtt{s}})]}{\textit{time}[\textit{exec } p\ (in_{\mathtt{s}}, in_{\mathtt{d}})] - \textit{time}[\textit{exec } p_{in_{\mathtt{s}}}\ (in_{\mathtt{d}})]} \quad [1.33]$$

Whether or not to favor specialization time (at the expense of the quality of the code produced) or the rapidity of the specialized program (at the expense of the specialization time) is a contextual choice; there is no better systematic strategy. It is the number of uses of a specialized program – generally unknown, but can be estimated by an expert or by using profiling – which guides the best choice. In practice, in terms of specializer engineering, the design choices made often lead us to prefer one strategy over another. However, a specializer may offer optimization options that slow down the production time, to the benefit of the quality of the specialized program produced.

In terms of an order of magnitude, again with Tempo, the break-even point for runtime specialization typically varies between 10 and 100 executions, with points at only three utilizations [NOË 98].

The same type of calculation may be carried out with program size. In terms of dynamic memory, account may have to be taken of whether the codes specialized during run time are stored in a specialization cache rather than systematically regenerated (see section 1.5.4) or erased and the corresponding memory space freed up.