Diss. ETH No. 10497

Code-Generation On-the-Fly:
A Key to Portable Software

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Michael Steffen Oliver Franz
Dipl. Informatik-Ing. ETH
born 1st May 1964
citizen of Germany

accepted on the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. J. Gutknecht, co-examiner

1994

Acknowledgement

I am deeply indebted to Professor N. Wirth for the privilege of being allowed to work with him. Over the years, he has taught me a wealth of wisdom that will follow me throughout my professional life, and beyond. I admire his unique ability of immediately recognizing the essential, and his courage to eliminate the superfluous even when considered indispensable by others. Often enough, I have attempted to apply his standards of judgement in my own work, and the results have been most gratifying. I thank him most sincerely for taking up the supervision of this thesis and commenting so competently on the ongoing project.

I thank Professor J. Gutknecht for his willingness to be my co-examiner and for advising me on this thesis. He is truly a great pedagogue, and if the contents of this thesis are comprehensible at all, this is also a result of his valuable criticism of an earlier draft, delivered in his unrivalled didactic style.

I am grateful to my colleagues at the Institut für Computersysteme for their openness to new ideas and their enthusiasm in discussing them, even late at night. Together, they have provided a friendly and intellectually most stimulating working environment.

Finally, I thank C. Pfister and S. Ludwig for proof-reading this thesis.

Contents

Abstract

A *technique for representing programs abstractly and independently of the eventual target architecture* is presented that yields a file representation twice as compact as machine code for a CISC processor. It forms the basis of an implementation, in which the process of code generation is deferred until the time of loading. At that point, native code is created on-the-fly by a *code-generating loader*.

The process of loading with dynamic code-generation is so fast that it requires little more time than the input of equivalent native code from a disk storage medium. This is predominantly due to the compactness of the abstract program representation, which allows to counterbalance the additional effort of code-generation by shorter input times. Since processor power is currently rising more rapidly than disk-access times and transfer rates are falling, the proposed technique is likely to become even more competitive as hardware technology evolves.

To users of the implemented system, working with modules in the abstract representation is as convenient as working with native object-files. Both kinds of file-representation coexist in the implemented system; they are completely interchangeable and modules in either representation can import from the other kind. Separate compilation of program modules with type-safe interfaces, and dynamic loading on a per-module basis are both fully supported.

Deferring code-generation until loading time can provide several new capabilities, especially when the intermediate program representation is machine-independent and thereby *portable*. It is argued that the combination of portability with practicality denotes an important step toward a *software-component industry*. Further benefits include a potential for reducing the number of recompilations after changes in source text, and a mechanism to decide at load time whether or not run-time integrity checks should be generated for a library module, eliminating the need to distinguish between development and production library configurations. All of these new possibilities have the potential of lowering the cost of software development and maintenance.

In the long run, fast on-the-fly code-generation may even replace *binary compatibility* for achieving software portability among processors implementing

the same architecture. Already today, different models of a processor family are diverging more and more and it is becoming increasingly difficult to serve all of them equally well with just one version of native code. If code is generated only at the time of loading, however, it can always be custom-tailored toward the specific processor that it will eventually run on.

Kurzfassung

Es wird eine *Technik zur abstrakten und Zielarchitektur-unabhängigen Programmrepräsentation* vorgestellt, die zu einem Dateiformat führt, welches doppelt so kompakt wie der Maschinencode eines CISC-Prozessors ist. Auf der Basis dieser Technik wurde ein System erstellt, in dem die Codegenerierung bis zum Zeitpunkt des Ladens verzögert wird. An jenem Punkt wird dann Maschinencode durch einen *codegenerierenden Lader* erzeugt.

Der Ladeprozess mit dynamischer Codegeneration ist annähernd so schnell wie das Lesen äquivalenten Maschinencodes von einem Plattenspeichermedium. Das ist hauptsächlich der Kompaktheit der abstrakten Programmrepräsentation zu verdanken, welche es gestattet, den Mehraufwand für die Codegenerierung durch einen reduzierten Eingabeaufwand auszugleichen. Vermutlich wird die vorgestellte Technik mit fortschreitender Hardwareentwicklung vergleichsweise sogar noch effizienter werden, da die Leistung von Mikroprozessoren momentan schneller wächst, als Plattenzugriffszeiten und -übertragungsraten abnehmen.

Für die Benutzer des erstellten Systems ist der Komfort bei der Verwendung von Modulen in der abstrakten Repräsentation vergleichbar mit demjenigen für herkömmliche Objektdateien. Die beiden Dateiformate können nebeneinander eingesetzt werden und sind vollständig gegeneinander austauschbar; auch können Module beider Repräsentationsformen einander wahlweise importieren. Die typsichere separate Compilation wird ebenso unterstützt wie das dynamische Laden auf Modulstufe.

Die Verzögerung der Codegeneration auf den Ladezeitpunkt birgt viele neue Möglichkeiten, besonders wenn die Programm-Zwischenrepräsentation maschinenunabhängig und damit *portabel* ist. In der vorliegenden Arbeit wird argumentiert, dass die Kombination von Portabilität und Einfachheit der Verwendung einen wichtigen Schritt hin zu einer *Industrie von Softwarebauteilen* bedeutet. Darüberhinaus wird die Voraussetzung dazu geschaffen, die Anzahl notwendiger Recompilationen nach Änderungen im Quelltext vermindern zu können. Auch wird ein Mechanismus eingeführt, der es gestattet, erst zur Ladezeit entscheiden zu müssen, ob ein Bibliotheksmodul Laufzeittests be-

inhalten soll oder nicht, womit die Notwendigkeit zur Unterscheidung von Entwicklungs- und Produktionskonfigurationen der Programmbibliotheken wegfällt. All diese neuen Möglichkeiten verkörpern ein Kostensenkungspotential für Softwareentwicklung und -unterhalt.

Langfristig ist sogar denkbar, dass schnelle dynamische Codegeneration anstelle von *Binärkompatibilität* eingesetzt werden wird, um Portabilität zwischen Prozessoren zu gewährleisten, die dieselbe Architektur implementieren. Schon heute unterscheiden sich die Modelle einer Prozessorfamilie immer stärker, und es wird immer schwieriger, mit einer einzigen Version von binärem Objektcode auf ihnen allen zufriedenstellende Leistungen zu erbringen. Wird der Maschinencode hingegen erst zur Ladezeit erzeugt, so kann er immer für denjenigen Prozessor massgeschneidert werden, auf dem er schlussendlich ausgeführt werden wird.

1. Introduction

The rapid evolution of hardware technology is constantly influencing software development, for better as well as for worse. On the downside, faster hardware can conceal the complexity and cost of badly-designed programs; Reiser [Rei89] is not far off the mark in observing that sometimes "software gets slower more quickly than hardware gets faster". On the other hand, improvements in hardware, such as larger memories and faster processors, have also provided the means for better software tools.

Often enough, methodical breakthroughs have been indirect consequences of better hardware. For instance, software-engineering techniques such as *information hiding* and *abstract data types* could be developed only after computers became powerful enough to support compilers for modular programming languages. Mechanisms such as *separate compilation* place certain demands on the underlying hardware and so had a chance of proliferation only after sufficiently capable computers were commonplace.

This thesis presents another example of a systematic technological advance that owes its viability to improved hardware. It describes a technique for representing programs abstractly in a format that is twice as compact as object code for a CISC processor. Combined with the speed of current processors and abundance of main storage, the use of an intermediate program representation based on the new technique makes it possible to accelerate the process of code generation to such a degree that it can be performed *on-the-fly* during program loading on ordinary desktop computers, even with a high resulting code quality.

A system has been implemented that permits the convenient use of program modules in this machine-independent intermediate representation, as if they had been compiled to native code. In this thesis, it is argued that such a system presents some completely new possibilities, among which lies the prospect of founding an industry offering *user-serviceable software components* that are immediately functional on more than one kind of target architecture.

Among the themes that recur throughout this thesis are *methods of program representation*, *code generation*, *modular programming*, and *software portability*.

This is a comparatively wide scope for a doctoral dissertation, which is reflected in a large and diverse list of references.

2. Program Representation

For every calculable function, there exists an **algorithm** that computes it [Chu35]. A *program* is an encoded description of such an algorithm that instructs a **universal computing machine** how to compute the corresponding function. A machine is called *universal* exactly if it can compute every function that is representable by an algorithm; Turing [Tur36] has shown that such universal machines exist. *Programming languages* correspond to abstract and relatively complex universal machines, while *digital computers* represent physical realizations of universal machines that are programmable in their characteristic machine languages.

Algorithms can be translated from one program representation into another; the mere existence of an algorithm guarantees that a corresponding program can be constructed for every machine that is universal. This chapter discusses the use of *intermediate languages* as transitional steps in such transformations between program representations, particularly solutions based on *abstract machines*. It then introduces a new program-representation technique called *semantic-dictionary encoding*, which achieves a high information density while facilitating simultaneously the efficient further translation into various machine languages.

Subdivided Compilers

Often, and for a variety of reasons, compilers are subdivided into smaller units. This may be a static separation into **modules**, or a dynamic partition into two or more separate **compilation phases** (or *passes*) that execute one after the other. The initial argument for such a division is often a physical limitation of the host machine that makes the construction of a monolithic compiler impossible. There are, however, other advantages to be gained from splitting a compiler into smaller parts, which make this approach attractive even when it is not mandated by external constraints, and keep it attractive when the constraints are eventually removed by evolving hardware technology.

Most importantly, subdividing a compiler usually reduces its complexity. The individual pieces are smaller and thereby simpler than the whole, and often the combined complexity of all parts taken together is less than that of a corresponding monolithic compiler. The reason for this is that the different parts can usually be decoupled from each other quite well, resulting in narrow interfaces between them. Complexity, on the other hand, is mostly rooted in non-local dependencies.

Another reason for structuring a compiler concerns the ease with which it can be retargeted for another architecture. By disentangling the code-generating functions from the rest of the compiler, the construction of a family of compilers for different target architectures can be simplified considerably. Adapting the compiler for a new target machine then amounts to the construction of a new **code generator**, while the rest of the compiler can remain unchanged. As a matter of fact, this technique has become so common that we tend to speak of "code generators" today as if they were self-contained programs in their own right, disregarding completely that syntactic analysis and code generation were not separated from each other in early compilers.

Intermediate Languages

A compiler transforms a program from one representation into another. When such a compiler consists of several phases, this implies the existence of further intermediate representations carrying the state of the compilation from one phase to the next. These intermediate representations are usually only transient data structures in memory, but in some compilers they have a linear form that can be stored on a data file. In the following, such stand-alone linear representations are referred to as **intermediate languages**.

Intermediate languages are interesting from the aspect of software portability. If an intermediate language is sufficiently simple, it will provide for the easy construction of a series of **interpreters** that can execute the intermediate language directly on a number of different target machines. Interpreters are usually easier to build than code generators, so that this approach is attractive, although it comes at the expense of reduced performance. Implementations that have been based on interpreted intermediate languages in this manner include *BCPL* [Ric71, RW79], *SNOBOL4* [Gri72], and *Pascal* [NAJ76].

Intermediate languages are often also used in the process of **bootstrapping** compilers onto new machines [Hal65]. Suppose that we have a *portable compiler* that translates a source language *SL* into an intermediate language *IL*, and that this compiler is itself written in *SL* and available encoded both in *SL* and in *IL*. Suppose further that we have an interpreter for *IL* that runs on a computer with a machine language *TL*. We may then modify the portable compiler (written in *SL*) to derive a native compiler that translates from *SL* to *TL* directly. By compiling it with the interpretable version of the original compiler we gain a native compiler that is written in *IL*, and hence executable by the interpreter. This compiler can now be used to create a native compiler capable of running directly on the target machine (Figure 2.1).

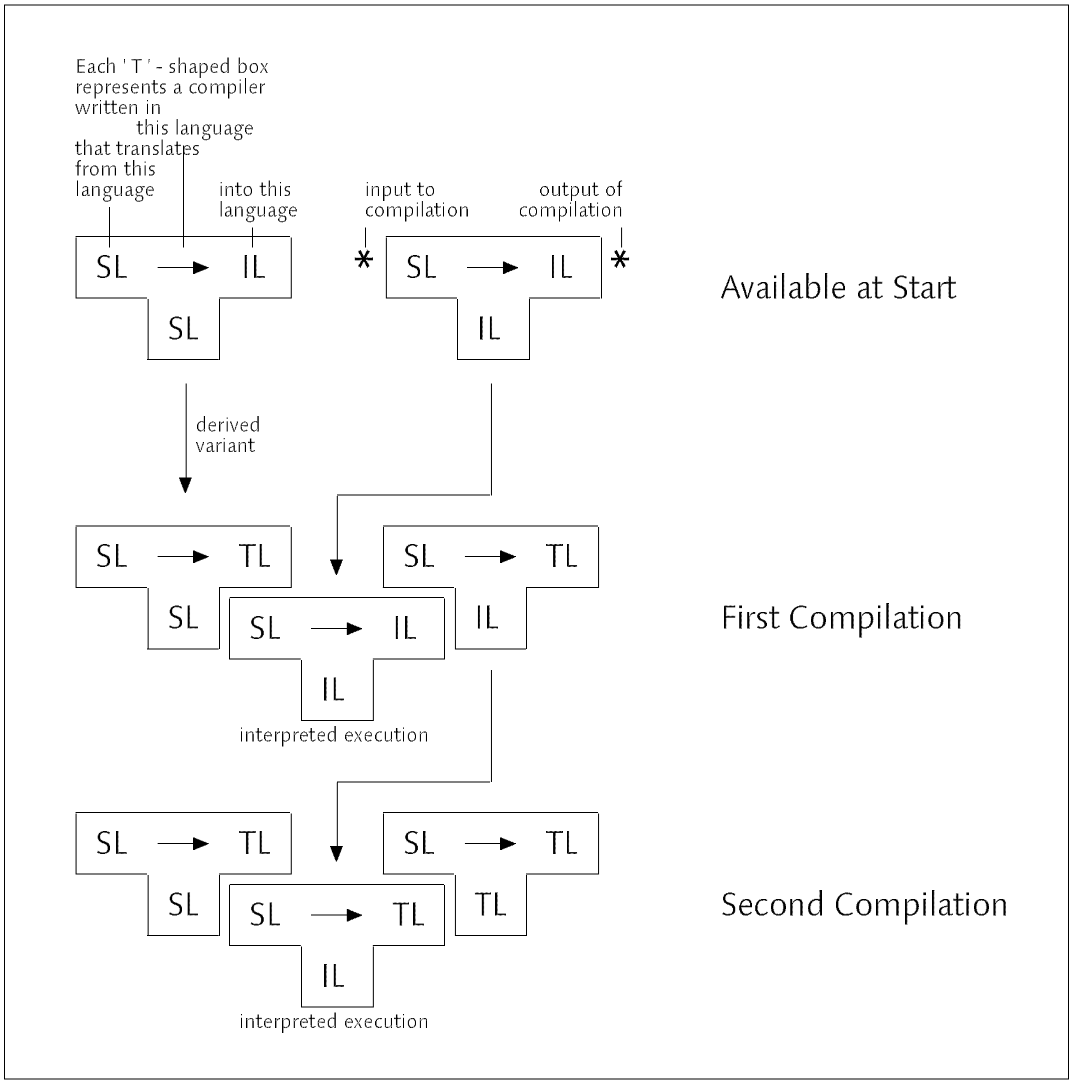


Figure 2.1: Bootstrapping a Compiler

UNCOL

In the 1950's, it became apparent that the ongoing proliferation of programming languages and hardware architectures would continue for some time. This would set off a vast demand for different compilers, as potentially one would require separate compilers for each combination of source language and target architecture.

To counteract the anticipated combinatorial explosion, the idea of a **linguistic switchbox** materialized in 1958 [SWT58, Con58, Ste60, Ste61a]. It was planned to design a *universal intermediate language*, into which programs originating in any problem-oriented language could be translated by an appropriate *generator* (*front-end* in today's terminology), and from which code for any processor architecture could be generated by a suitable *translator* (*back-end*). This would enable the construction of compilers for *m* languages to be run on *n* machines by having to write only *m+n* programs (*m* front-ends and *n* back-ends), instead of *m∗n* (Figure 2.2). The intermediate language was to be called *UNCOL*, for *u*niversal *c*omputer-*o*riented *l*anguage.

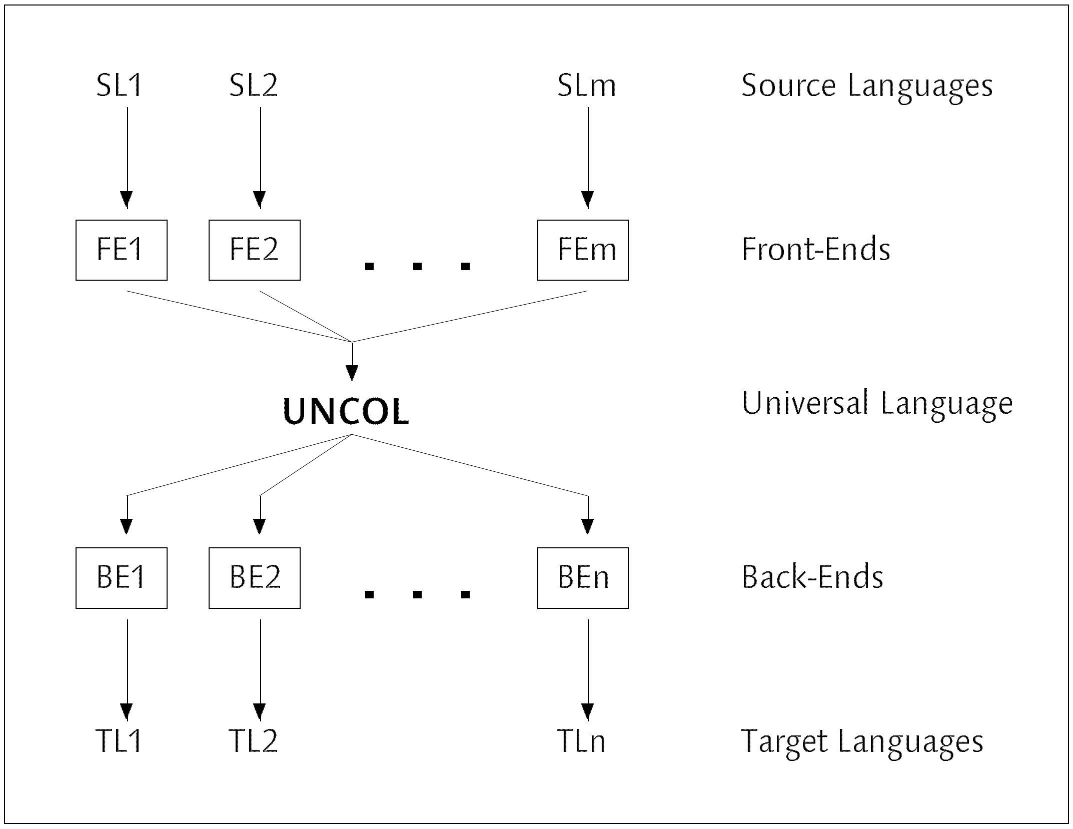


Figure 2.2: The Use of UNCOL as a Linguistic Switchbox

One might of course wonder why such an intermediate language needs to be *designed specifically* for this purpose. If algorithms can be translated from one program representation into another, then in principle the machine language of any sufficiently powerful computer could serve as UNCOL. Unfortunately, however, most translations between languages are so difficult to do that they are not practical. To wit, potential UNCOL candidates need to be amenable to efficient implementation. They should also provide for further developments in source and target languages. Steel [Ste61b] cites the example of a hypothetical UNCOL specified before the invention of *index registers*, which would be very cumbersome to use with programs operating on arrays.

To this date, no proposed UNCOL has been met with universal agreement. However, many **compiler families** have offered varying combinations of front-ends and back-ends by way of common intermediate representations, and thereby kept the spirit of UNCOL alive. Moreover, certain programming languages have been implemented on so many different architectures that they have attained an almost UNCOL-like status. For instance, the programming language *C* [KR78] is nearly pervasive. Atkinson et al. [ADH89] describe a *Cedar* compiler that achieves portability by performing high-level **source-to-source translation**, producing C as its output. The resulting C programs are then passed directly to a C compiler, without further human editing. They serve only as an intermediate representation in this context.

Abstract Machines

A straightforward method for obtaining an intermediate-language representation of a program is to represent it as an instruction sequence for some *fictitious computer*, also called an **abstract machine** [PW69, NPW72, KKM80]. Most intermediate languages, including the very first UNCOL ever proposed [Con58], follow this pattern. Besides abstract machines, intermediate languages have also been based on linearized parse-trees [GF84, DRA93a]. Furthermore, there are compilers that compile via ordinary high-level programming languages by way of source-to-source program translation as mentioned above [ADH89].

Abstract machines reached their heyday in the early 1970's. They were widely popular for a decade, providing software portability to a variety of target architectures while consuming only moderate resources. There were even architectures that started out as abstract machines and, because of their popularity, were subsequently realized physically [WD79] or implemented in microcode [Hal82]. Some of the better known abstract machines from that era are the following:

- *BCPL*. BCPL [Ric69] is a small, block-structured programming language aimed primarily at systems-programming applications. It has only one data type, the *Rvalue*, to which the language attaches different interpretations depending on the operation applied. Each Rvalue occupies one unit of store on an idealized object computer that is visible at the level of the source language. The implementation of BCPL [Ric71, RW79] is based on an *abstract stack machine*. BCPL's type-less semantic model of an idealized computer allows for portable address arithmetic in source programs and target-machine-independent stack management in the compiler.

- *SNOBOL4*. SNOBOL4 is a member of the *SNOBOL* family of character-string-processing languages [Gri78]. Its original implementation [Gri72] uses a *register-less abstract machine* that offers only memory-to-memory instructions. The basic data unit of this abstract machine is called a *descriptor*. All of SNOBOL4's data types are encoded within such descriptors. However, the internal layout of descriptors varies between different target architectures, allowing for an efficient implementation of the interpreter on different kinds of hardware, but requiring changes in the SNOBOL4 compiler whenever the system is ported.

- *Pascal-P*. The general-purpose programming language Pascal [Wir71] is still widely used today. One of its early implementations, *Pascal-P* [NAJ76], is founded on a hypothetical stack computer similar to the abstract machine used in the implementation of BCPL. Unlike BCPL, however, Pascal differentiates between various data types. In the Pascal-P implementation, the abstract-machine representations of these types are adapted to each individual target architecture, with respect to their storage requirements, and to their alignment. As in the implementation of SNOBOL4, this target-architecture dependence prevents direct portability of the intermediate language and requires parametrization of the compiler, but results in efficient storage allocation and execution of the interpreter.

- *Janus*. Unlike the previous three examples, each of which uses an abstract machine to implement a single source language on a variety of target architectures, *Janus* [CPW74, HW78] comes closer to the original idea of UNCOL by defining the essentials of an intermediate language independently of any source language or target architecture. Rather than providing a single abstract machine, Janus describes the *common basic structure* (stack machine with index register) of a whole *family of abstract machines*, which differ in their instruction sets to accommodate the specific constructs of source languages. Janus has been used successfully in the development of compilers for the programming languages Pascal [Wir71] and Algol-68 [WMP69].

The advent of the *personal computer* eventually caused the demise of the abstract machine, since it led to a de-facto standardization of the machine language on the low end of the computer performance spectrum while simultaneously making these computers affordable. In the light of a quasi-standard machine language, however, there was no longer a need to use intermediate languages to achieve software portability, even more so as the interpreted execution of abstract machines was associated with inefficiency.

This picture is changing again only now. Although the standardization of hardware has continued over the past two decades, leaving only about a dozen major architectures to which software needs to be ported, the appeal of traditional *binary compatibility* is waning. This is because different implementations of the same architecture begin to diverge by so much that it is becoming more and more difficult to generate native code that performs well on all processors within a family. As will be shown in a later chapter, the work presented in this thesis offers an alternative, by allowing to generate code quickly at the time of loading. It is thereby possible to deliver optimized instruction sequences to individual processors while still maintaining a user-convenience comparable to that of binary compatibility.

Disadvantages of Abstract Machines

Abstract machines are often defined by forming a conceptual intersection of potential target-machine architectures. Hence, intermediate languages based on abstract machines are by their very nature closer to machine languages than to high-level programming languages. Unfortunately, this has the undesirable consequence that the abstract-machine representation of an algorithm contains *less structure* than the corresponding source program. Most abstract machines can describe only the algorithmic aspects of a computation, and cannot correctly preserve all of the higher-order information expressed in high-level-language programs, such as block structure and data typing.

For example, consider the following source fragment in the programming language Oberon [Wir88]:

```
VAR
   ch: CHAR;
   lint: LONGINT;
BEGIN
   lint := LONG(ORD(ch));
```

A possible representation as an instruction sequence for an abstract stack computer might be the following:

```
LOAD1    ch          move single-byte CHAR value onto stack
ORD                  zero-extend, result is an INTEGER
LONG                 sign-extend, result is a LONGINT
STORE4   lint        move four-byte LONGINT value back to memory
```

However, this sequence of operations can be performed by a single instruction on certain processors, such as those of the *National Semiconductor 32000 family* [NS84]:

```
MOVZBD  ch, lint     move, zero-extending byte to double-word
```

Hence, it may be necessary to combine several abstract-machine instructions into one target-machine instruction when generating anything but the most naive code out of an abstract-machine intermediate representation. In these cases, the code generator effectively needs to reconstruct, at considerable expense, information that was more easily accessible in the front-end, but lost in the transition to the intermediate representation.

The only way to solve this particular problem is by bridging the *semantic gap* through the introduction, on the level of the abstract machine, of a *separate move instruction for every combination of source and destination data type*. Still, this is able to rectify only those cases that involve a direct data-format conversion. As target architectures may potentially offer arbitrarily complex operations, e.g., combined move-and-shift operations, the merger of several abstract-machine instructions into one single target-machine instruction can never be ruled out completely, no matter how elaborate the abstract machine's design.

Because of their systematic defects, intermediate languages based on abstract machines are not well suited as a basis for the fast generation of high-quality native code. The remainder of this chapter introduces an intermediate representation that is. This representation, which I call **semantic-dictionary encoding** (**SDE**), preserves the full semantic context of source programs while being highly compact.

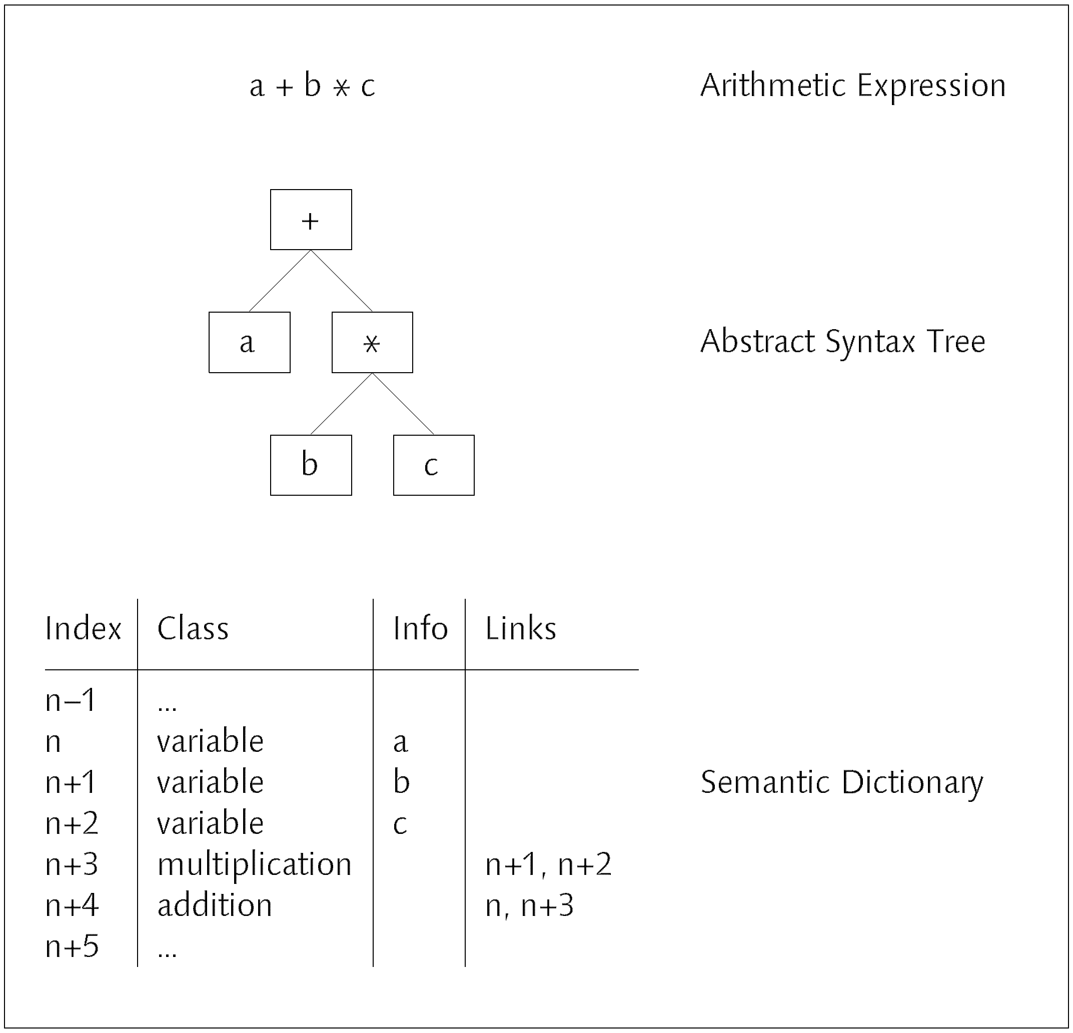
An Overview of Semantic-Dictionary Encoding

SDE is a dense representation. It encodes a syntactically correct source program by a succession of indices into a **semantic dictionary**, which in turn contains the information necessary for generating native code. The dictionary itself is not part of the SDE representation, but is constructed dynamically during the translation of a source program to SDE form, and reconstructed before (or during) the decoding process. This method bears some resemblance to commonly used data compression schemes [Wel84].

With the exception of wholly constant expressions, which are evaluated during the transformation of a source program to SDE form, SDE preserves all of the information that is available at the level of the source language. Hence, unlike abstract-machine representations, transformation to SDE preserves the *block structure* of programs, as well as the *type* of every expression. Moreover, when used as the input for code-generation, SDE in certain cases provides for *short-cuts* that can increase translation efficiency.

A program in SDE form consists of a **symbol table** (in a compact format, as explained in [Fra93b]) and a series of **dictionary indices**. The symbol table describes the names and the internal structure of various entities that are referenced within the program, such as *variables*, *procedures*, and *data types*. It is used in an initialization phase, in the course of which several *initial entries* are placed into the semantic dictionary. The encoding of a program's actions consists of references to these initial dictionary entries, as well as to other entries added later in the encoding process.

Dictionary entries represent nodes in a directed acyclic graph that describes the semantic actions of a program abstractly. In its most elementary form, a semantic dictionary is simply such an **abstract syntax-tree** in tabular shape, in which the references between nodes have been replaced by table indices. Each dictionary entry consists of a *class* attribute denoting the semantic action that the entry stands for (e.g., *assignment*, *addition*, etc.), and possibly some references to objects in the symbol table (connected via an *info* attribute in the following diagram) and to other dictionary entries (described by the *links* attribute). Figure 2.3 gives an example of a simple arithmetic expression represented as an abstract syntax tree and as a semantic dictionary.



Index	Class	Info	Links
n–1	...		
n	variable	a	
n+1	variable	b	
n+2	variable	c	
n+3	multiplication		n+1, n+2
n+4	addition		n, n+3
n+5	...		

Figure 2.3: Different Representations of an Expression

What differentiates a tabular abstract syntax-tree from a semantic dictionary is that the latter can describe also **generic characteristics** of *potential nodes* that might appear in such a tree. In addition to *complete entries* that directly correspond to nodes of an abstract syntax tree, semantic dictionaries contain also *generic*, or **template entries**. These templates have the same structure as complete entries, but at least one of their attributes is missing, as recorded in a status flag. In SDE, complex expressions can be represented not only by complete entries, but also by templates in combination with other entries. For example, the expression " a + b " can be represented by the index of an "addition" template followed by the indices of two variable-reference entries.

Now suppose that a template existed in the semantic dictionary for every construct of the source language (*assignment*, *addition*, etc.), and that furthermore the semantic dictionary were initialized in such a way that it contained at least one entry (possibly a template) for every potential use of every object in the symbol table. For example, an object describing a *procedure* in the programming language Oberon [Wir88] requires a minimum of four entries in the dictionary, relating in turn to a *call* of the procedure, *entry* of the procedure, *return* from the procedure, and *addressing* the procedure, which is used in the assignment of the procedure to procedure variables. There are no other operations involving procedures in Oberon.

These preconditions can be fulfilled by initializing the semantic dictionary in a suitable way. They enable us to represent any program by only a symbol table and a succession of dictionary indices. As an example, consider the following module *M* in the programming language Oberon:

```
MODULE M;

   VAR i, j, k: INTEGER;

   PROCEDURE P(x: INTEGER): INTEGER;
   BEGIN  ...
   END P;

BEGIN
   i := P(i);   i := i + j;   j := i + k;   k := i + j;   i := i + j
END M.
```

In order to encode this program by the method of SDE, we first need to initialize the semantic dictionary, which depends on the operations offered by the programming language (Oberon in this case) and on the objects in the symbol table. The symbol table for the example program contains three integer variables (*i*, *j*, and *k*) and a function procedure (*P*). After initialization, the corresponding semantic dictionary might look like the following (the individual entries' indices are represented by symbolic names so that they can be referenced further along in this chapter, and missing attributes of templates are denoted by a dot).

Index	Class	Meaning	Missing
asgn	assignment	. := .	left, right
plus	addition	. + .	left, right
...	...	...	...
vi	variable	i	–
vj	variable	j	–
vk	variable	k	–
refp	address	P	–
callp	function call	P(.)	left
entp	entry	P-BEGIN	–
retp	return	P-RETURN .	left
...	...	...	...

The instruction sequence that constitutes the body of module *M* may then be represented by the following sequence of 24 dictionary indices:

asgn vi callp vi	$i := P(i)$
asgn vi plus vi vj	$i := i + j$
asgn vj plus vi vk	$j := i + k$
asgn vk plus vi vj	$k := i + j$
asgn vi plus vi vj	$i := i + j$

This is where the second major idea of SDE comes in. What if we were to keep on *adding* entries to the dictionary during the encoding process, based on the expressions being encoded, in the hope that a *similar expression* would occur again later in the encoding process? For example, once we have encoded the assignment " i := P(i) " we might add the following three entries to the dictionary:

Class	Meaning	Missing
function call	P(i)	–
assignment	i := .	right
assignment	i := P(i)	–

Thereafter, if the same assignment " i := P(i) " occurs again in the source text, we can represent it by a single dictionary index. If another assignment of a different expression to the variable *i* is come across, this may be represented using the template "assign to *i*", resulting in a shorter encoding.

Encoding module *M* in this manner results in the following additional entries being placed in the dictionary (assuming that, after initialization, the dictionary comprised *n-1* entries):

Index	Class	Meaning	Missing
...	...	...	...
n	function call	P(i)	–
n+1	assignment	i := .	right
n+2	assignment	i := P(i)	–
n+3	addition	i + .	right
n+4	addition	. + j	left
n+5	addition	i + j	–
n+6	assignment	i := i + j	–
n+7	addition	. + k	left
n+8	addition	i + k	–
n+9	assignment	j := .	right
n+10	assignment	j := i + k	–
n+11	assignment	k := .	right
n+12	assignment	k := i + j	–
...	...	...	...

The body of module *M* can then be encoded as follows:

```
asgn  vi   callp  vi                                   i := P(i)
n+1        plus  vi  vj                                i := i + j
asgn  vj   n+3  vk                                     j := i + k
asgn  vk   n+5                                         k := i + j
n+6                                                    i := i + j
```

Instead of the previous 24 dictionary indices, this encoding requires only 16 of them, although the individual indices themselves may be larger since the dictionary has more entries. Still, the second encoding is usually much more space-economical and has further advantages, as will be shown in the following paragraph.

Increasing the Speed of Code Generation

The decoding of a program in SDE form is similar to the encoding operation. At first, the dictionary is initialized to a state identical to that at the onset of encoding. Since the symbol table is part of the SDE file-representation, the decoder has all required information available to perform this task.

Thereafter, the decoder repeatedly reads dictionary indices from the file, looking up each corresponding entry in the semantic dictionary. Whenever a complete entry is found in this manner, its meaning is encoded directly in the dictionary and the decoder can proceed to process the next index on the input stream. If a template is retrieved instead, it is copied, further entries corresponding to its undefined attributes are input in turn, and the modified copy is added to the dictionary as a new complete entry. Moreover, additional templates are sometimes added to the semantic dictionary according to some fixed heuristics, in the hope that a corresponding branch of the program's syntax tree will show up further along. A more detailed discussion of these heuristics follows in the next chapter.

In addition to providing a dense program representation, SDE is able to supply certain information that is not explicitly available in source text, namely about **multiple textual occurrences of identical subexpressions** (including, incidentally, *common designators*). This can sometimes be exploited during code generation, resulting in an increase in the speed of code output.

Consider again the (second, more compact) SDE-representation of module *M* above. Now suppose that we want to generate object code for a simple stack machine directly from this SDE form. Let us assume that we have processed the first two statements of *M*'s body already, yielding the following instruction sequence starting at address *a*:

```
a      LOAD   i          load i onto stack
a+1    BRANCH P          call procedure P with argument i
a+2    STORE  i          assign result to i
a+3    LOAD   i          load i
a+4    LOAD   j          load j
a+5    ADD               add i to j
a+6    STORE  i          assign their sum to i
```

Let us further assume that we have kept a note in the decoder's semantic dictionary, describing which of the generated instructions correspond to what dictionary entry. For example, we might simply have recorded the program counter value twice for every entry in the semantic dictionary, both before and after generating code for the entry:

Index	Class	Meaning	Begin	End	Invar
...	...	...	...	...	...
n	function call	P(i)	a	a+1	No
n+1	assignment	i := P(i)	a	a+2	No
...	...	...	...	...	...
n+5	addition	i + j	a+3	a+5	Yes
n+6	assignment	i := i + j	a+3	a+6	Yes
...	...	...	...	...	...

A setup such as this allows us to bypass the usual code generation process under certain circumstances, replacing it with a simple *copy operation* of instructions already generated. For example, when we encounter the second reference to entry *n+6* in the SDE-representation of module *M*, we know that we have already compiled the corresponding statement " i := i + j " earlier on. In this case, we may simply re-issue the sequence of instructions generated at that earlier time, which can be found in the object code between the addresses *a+3* and *a+6*, as recorded in the dictionary.

Of course, this method cannot be applied in every case and on all kinds of machine. First of all, code-copying is possible only for **position invariant** instruction sequences. For example, a subexpression that includes a call of a local function by way of a *relative branch* is not position invariant because the branch distance is different each time that the function is called. It happens that information about position invariance can also be recorded conveniently in the dictionary, as shown in the example above.

Secondly, the instruction sequences obtained by code-copying may not be optimal for more complex processors with many registers and multistage instruction-pipelines. For these machines, it may be necessary to employ specific optimization techniques. However, since SDE preserves all information that is available in the source text, arbitrary optimization levels are possible at the time of code generation, albeit without the shortcuts provided by code-copying. One then simply treats the semantic dictionary as an abstract syntax-tree in tabular form.

The interesting part is that code-copying is beneficial especially on machines that are otherwise slow, i.e., CISC processors with few registers. Consider module *M* again, in which the subexpression " i := i + j " appears three times. On a simple machine that has only a single accumulator or an expression stack, the identical instruction sequence *will have to be used* in each occurrence of the subexpression. Code-copying can accelerate the code-generation process in this case. The optimal solution for a modern RISC processor, on the other hand, might well consist of three distinct instruction sequences for the three instances of the subexpression, due to the use of register variables and the effects of instruction pipelining. However, such processors will generally be much faster, counterbalancing the increased code-generation effort, so that an acceptable speed of code generation can still be achieved even if optimization is necessary.

3. The SDE Encoder and Decoder

The previous chapter has introduced the technique of *semantic-dictionary encoding* (SDE). On the basis of this encoding technique, a system has been implemented, in which the code generator is no longer part of the compiler, but has been incorporated into the module loader. The implemented system features an **Oberon-to-SDE compiler** that transforms programs written in the programming language Oberon [Wir88] into the SDE file-representation, and a **code-generating loader** that reads SDE-files and uses the information contained in them for generating code on-the-fly for *Motorola 68020* processors [Mot87].

This chapter discusses the encoding and decoding mechanisms and relates the motivation behind key design decisions. Most importantly, it explains some of the heuristics used in the management of semantic dictionaries.

Making use of Scoping

Many programming languages limit the scope of certain objects appearing within a program, for example by restricting the visibility of a procedure's local variables to the extent of the procedure body. These **scoping rules** can be exploited for reducing the size of the semantic dictionary. Smallness of the semantic dictionary is beneficial because a program in SDE form is composed of *dictionary indices*. The larger the dictionary, the more bits are required on average for the representation of these indices.

It is therefore desirable to remove from the semantic dictionary continuously all entries which (directly or indirectly) relate to objects that have become inaccessible by the scoping rules. However, the encoder and the decoder of the SDE format need to synchronize their dictionary operations. Any removal of an entry from the encoder's dictionary must, therefore, be communicated to the decoder, and this can happen only via the encoded program itself.

We also note that there are objects that are visible throughout a module, such as *constants* and *global variables*. Furthermore, the dictionary contains also the *operation templates* that are used for encoding the semantic actions of a program. These must never be removed from the dictionary.

An easy and efficient solution to dictionary management, taking into account these differences between temporary and permanent entries, is to let the dictionary grow *in two directions* simultaneously, as illustrated in Figure 3.1. Entries with unlimited scope are added at one end of the dictionary, which grows unboundedly. All other nodes are added at the opposite end of the dictionary, which is managed as a stack. Every time a new scope is opened, the dictionary's current extent in this second direction is marked. It is reset to the same point when the scope is closed again. The decoder of the SDE format can keep track of these mark and restore operations by monitoring its input stream for the occurrence of dictionary indices denoting *procedure entry* and *return*.

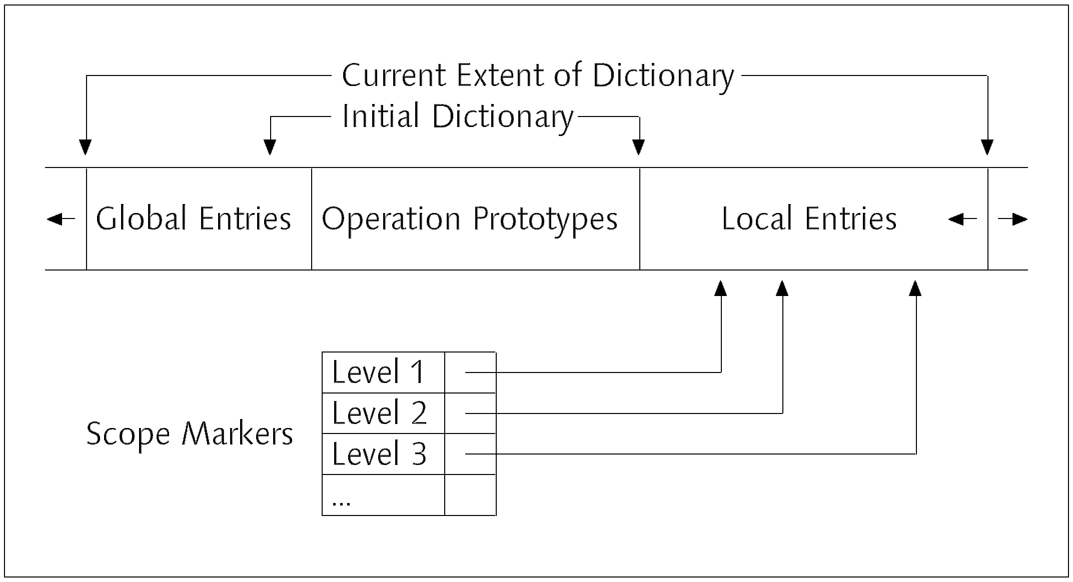


Figure 3.1: Two Directions of Dictionary Growth

Representing Indices on the File

In the implemented system, dictionary indices are represented on SDE-files in a **variable-length data format** based on a *stop-bit*, as has been described in [Fra93b]. This variable-length data format requires less space for encoding small absolute values than it needs for encoding large absolute values.

Such a compact format is ideally suited for representing dictionary indices. However, the information density of the resulting encoding depends significantly on the magnitudes of the individual indices and, therefore, on the distance of entries from the origin of the dictionary. For a compact encoding, it is essential to *map the index vector* onto the dictionary in such a way that often-occurring dictionary entries come to lie within the range that can be addressed by a minimum number of bits.

Experiments have revealed that the most common dictionary references in SDE-encoded programs relate to templates describing the primitive operations of the source language, and to expressions involving local variables. A dictionary addressing scheme as indicated in Figure 3.2 has, therefore, been adopted. Since most procedures start at lexical level zero and are relatively short (i.e., they generate few new dictionary entries), a high proportion of the dictionary indices appearing in typical SDE representations will then actually fall within the range of indices that can be represented space-efficiently.

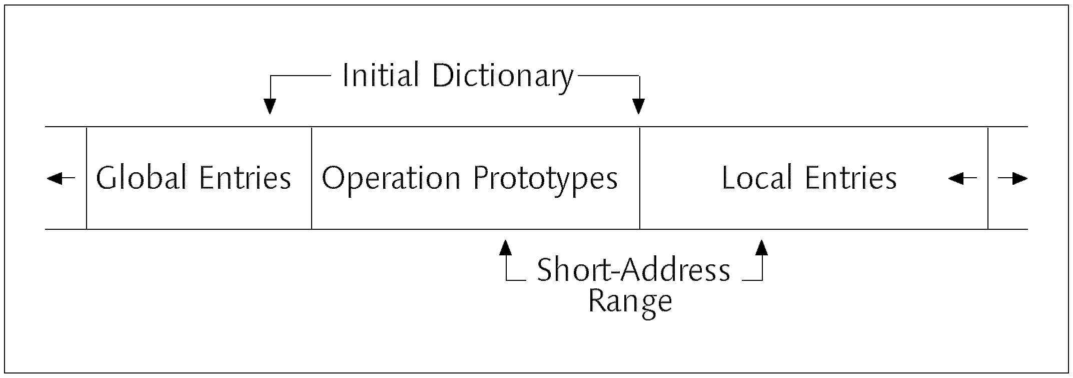


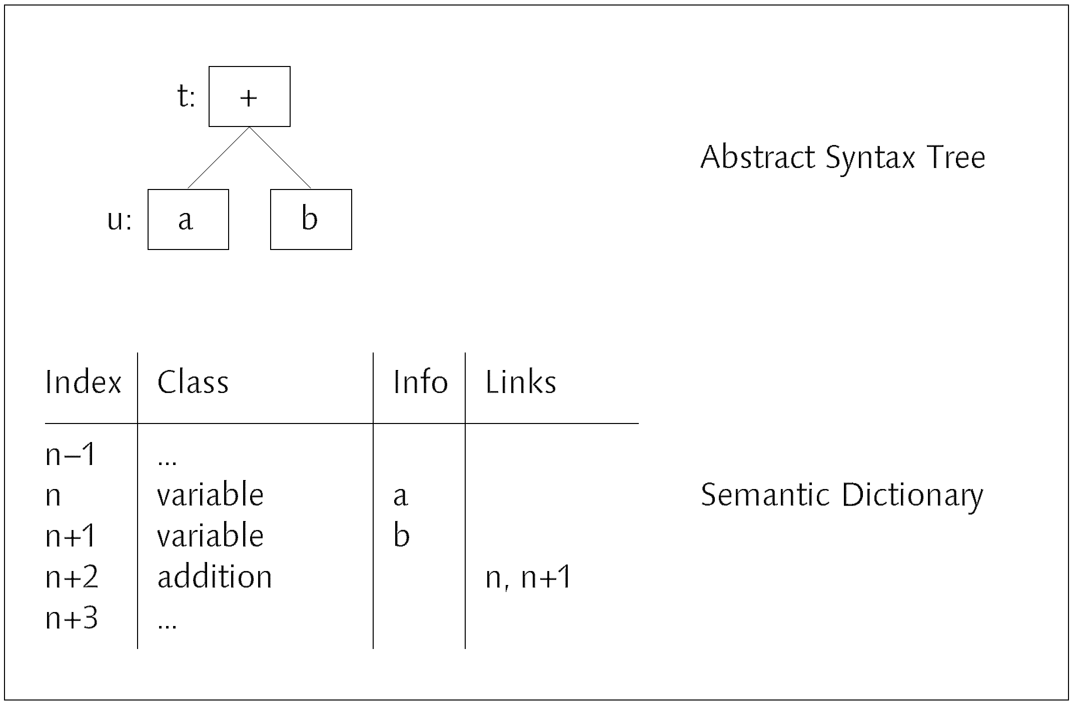
Figure 3.2: Relative Position of the Short Dictionary Addressing Range

Encoding Oberon Source-Programs into SDE

The **Oberon-to-SDE compiler** incorporates a fairly conventional compiler front-end [Cre91]. It parses each Oberon source text by recursive descent and builds a symbol table as well as an **abstract syntax tree**, i.e., a directed acyclic graph that describes the semantic actions of the program. If no error is detected during parsing, control passes to the **encoder**, which creates an SDE-file representing the contents of the source program in a compact and machine-independent manner.

In doing so, the encoder *traverses* the abstract syntax tree. For each node reached during this traversal, it searches the current semantic dictionary for an entry with a high **conformance** to the node. Conformance describes how well a dictionary entry matches a node of the tree. An entry is **fully conforming**, or

isomorphic, to a node of the abstract syntax tree, if both of them describe the same semantic action, and if they both either have no descendants at all, or if all of their corresponding descendants are isomorphic to each other. For example, in Figure 3.3, the dictionary entry with index *n+2* is isomorphic to the node labelled *"t"*, and the entry with index *n* is isomorphic to node *"u"*.

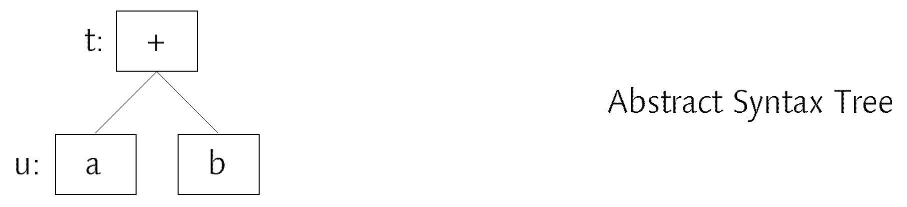


Index	Class	Info	Links
n–1	...		
n	variable	a	
n+1	variable	b	
n+2	addition		n, n+1
n+3	...		

Figure 3.3: Dictionary Entry "n+2" *is Isomorphic to Node* "t".

Only complete entries (i.e., entries with no undefined descendants) can conform fully to syntax-tree nodes. Conformance for templates is defined in a more modest manner. A template conforms to a node of the abstract syntax-tree, if it describes the same semantic action and all of its *defined descendants* are isomorphic to the corresponding descendants of the node. For example, in Figure 3.4 below, the templates with indices *x*, *x+1*, and *x+2* all conform to the node labelled *"t"*.

The search for a conforming dictionary entry succeeds always, because through the initialization of the dictionary we guarantee that there is at least one minimally conforming template in the dictionary for every possible node in the abstract syntax tree. After finding a conforming dictionary entry, the encoder writes its index to the SDE-file and then traverses recursively all of the sub-trees of the abstract syntax-tree that correspond to undefined descendants of the conforming entry. In cases in which there are several alternative entries that all conform to a node in the abstract syntax tree, as in the example of Figure 3.4, the encoder can maximize the information density of the SDE-file by selecting always the entry with the largest number of defined descendants.



Index	Class	Info	Links
n–1	...		
n	variable	a	
n+1	variable	b	
...	...		
x	addition		. , .
x+1	addition		n, .
x+2	addition		. , n+1

Semantic Dictionary

Figure 3.4: Dictionary Entries "x", "x+1", *and* "x+2" *all Conform to Node* "t".

The representation used in SDE-files is heavily biased towards the ease of *decoding*. The process of *encoding* requires many searches in the dictionary and is, therefore, inherently less efficient than decoding. In principle, it would be quite possible to perform all of these required searches by traversing the whole dictionary linearly each time until an adequate entry had been found. However, this would make the semantic-dictionary encoding of large programs unpractical.

Instead of searching the whole dictionary over and over, the current implementation uses an overlay sorting structure to reduce the number of dictionary entries that need to be inspected. Dictionary entries are grouped by their semantic action in this sorting structure, and all entries related to a certain procedure-scope are removed as soon as that scope has been closed.

Note that for correctness of the algorithm, it would suffice to reset the dictionary *counter* at the end of each scope, while the entries themselves would not have to be removed from the dictionary. Due to the nature of the scoping rules, it would be impossible for any dictionary entry ever to be returned as the result of a search after the corresponding scope had been closed.

Heuristics of Dictionary Management

There are several time and space trade-offs to be considered in the algorithm that manages the semantic dictionary. This becomes most important when considering the strategy by which new templates are added. In general, adding a greater number of templates leads to a denser program representation on SDE-files, but it also inflates the size of the dictionary and thereby the memory requirements of the encoding and decoding steps. It may also increase the time of decoding, due to the overhead of having to allocate a larger dictionary and having to perform more initializations. The following discussion should exemplify the sort of questions that need to be answered in the design of dictionary-management heuristics. As before, an expression syntax is used, in which a dot stands for a missing link of a template entry.

Consider the expression " a + b ". Since *addition* is a fundamental operation of the Oberon language, a template for the generic expression " . + . " is guaranteed to exist in every Oberon-specific semantic dictionary after its initialization. Similarly, if some variables " a " and " b " appear in the symbol table of a certain program, then the corresponding semantic dictionary will be initialized automatically to contain appropriate references. The expression " a + b " occurring at the very beginning of a program will, therefore, be translated into a sequence of three dictionary indices, standing for " . + . ", " a ", and " b ".

At this point, the designer of the dictionary-management heuristics has *a choice* which of the related entries " . + b ", " a + . ", and " a + b " should be added to the dictionary. Clearly, the strategy that yields the most compact SDE-files, at the expense of table space, is to add all three variants. For the moment, let us assume that all three variants were added. Now consider what happens if, further down in the same program, the expression " a + x " is come across. This can be encoded based on the entry " a + . " already in the semantic dictionary. But then what? Again there is a choice of further entries that could be added to the dictionary. Clearly, adding " a + x " is beneficial, but what about " . + x "?

The problem lies in the fact that " . + x " may very well be in the semantic dictionary already, for example, resulting from a previous encoding of " y + x ". However, determining whether an entry is in the dictionary requires a search, which is expensive and, therefore, not acceptable at the the time of decoding. Hence, there is a choice of either leaving the entry in question out of the dictionary altogether, or to blindly enter it anyway, possibly for a second time and at a waste of table space.

The question of which strategy is better is still very much open. Different heuristics of template-generation have been experimented with, leading to the observation that the effect varies with the style in which the original source program has been written, depending, among other things, on the number of common subexpressions that have been factored out by the programmer explicitly. A possible solution would of course be to try several variants during the encoding process, finally using the strategy best suited for each particular source program. The corresponding heuristic to be applied during decoding could then be identified by a tag in the SDE-file. To keep it simple, the current implementation uses a straightforward strategy generating relatively few templates.

Factorization of Procedure Calls

When we examine existing Oberon programs, we observe that the parameter lists of procedures are usually arranged in such a way that parameters become "more variable" to the right side of a parameter list. For example, the procedure

```
WriteBytes(VAR r: Files.Rider; VAR x: ARRAY OF CHAR; n: LONGINT)
```

is likely to be called several times for any fixed rider *r* with varying further parameters. Moreover, the actual parameter substituted for *n* is still more likely to vary than the one assigned to the formal parameter *x*.

This arrangement is partly due to the extensive use of abstract data types, which naturally appear as the first parameters of the procedures operating on them. However, there also seems to exist a deeper, unwritten, but generally observed rule among programmers to structure parameter lists in the way mentioned. The dictionary-management heuristics used in the encoding of procedure calls has been based on the hypothesis that it is natural to place "less variable" parameters before "more variable" ones.

Consequently, the implemented system supports the factorization of common parameter lists that partially match from the left side. For example, after encoding the procedure call " P(x, y, z) ", a corresponding complete entry is added to the semantic dictionary, as are the two templates " P(x, ., .) ", and " P(x, y, .) ". A subsequent procedure call " P(x, y, a) " can then be represented space-efficiently by only two dictionary indices, standing for " P(x, y, .) " and " a ".

This strategy represents a trade-off between information density and size of the dictionary, while it is amenable to efficient implementation. An adequate mapping onto a semantic-dictionary representation can be found quite simply by viewing each procedure call as a *binary tree of partial evaluations*, as shown in Figure 3.5.

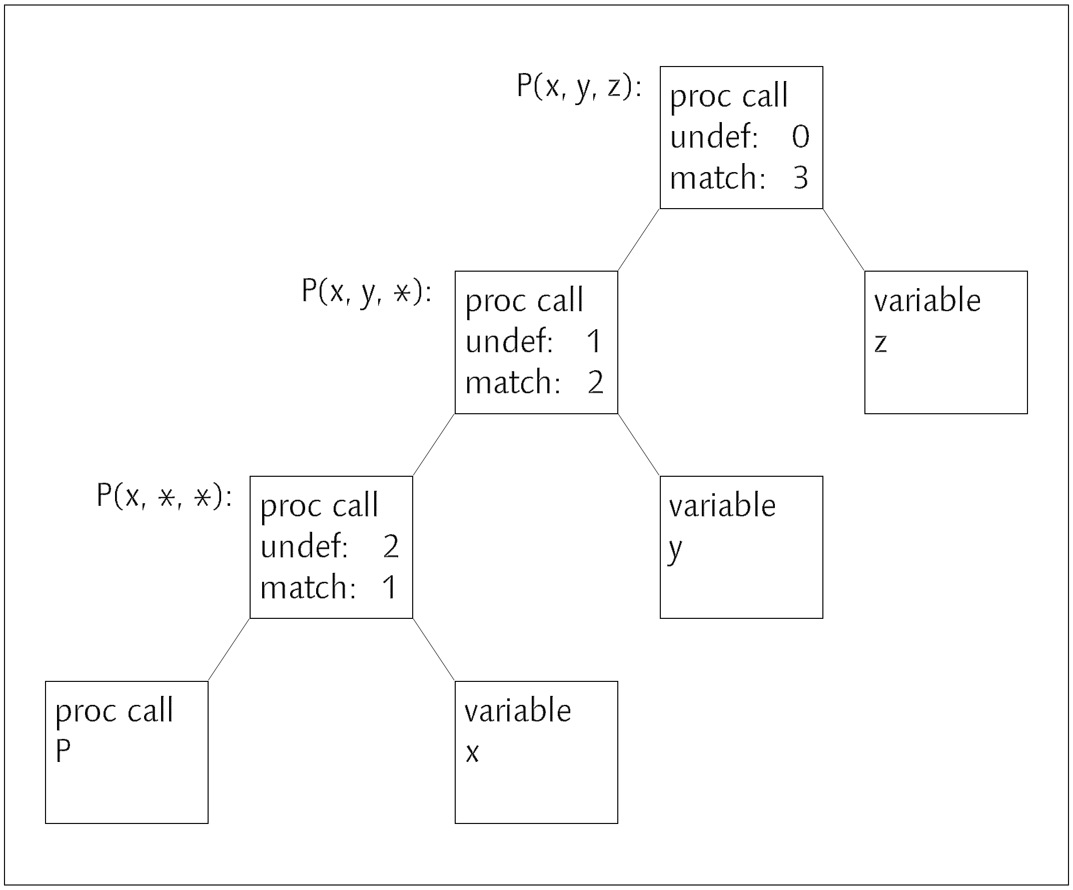

P(x, y, z):
proc call undef: 0 match: 3
P(x, y, *):
proc call undef: 1 match: 2
variable z
P(x, *, *):
proc call undef: 2 match: 1
variable y
proc call P
variable x

Figure 3.5: Tree View of a Procedure Call.

An alternative strategy would have been to generate even further dictionary entries describing every possible permutation of *wild-cards* in parameter lists. For example, after encoding the procedure call " P(x, y, z) " above, one might have also included the following entries, in addition to the ones mentioned previously: " P(., y, .) ", " P(., ., z) ", " P(., y, z) ", and " P(x, ., z) ".

Preparing the Dictionary for Decoding

In the implemented system, SDE-files are decoded at the time of loading. This is done by an SDE-decoding code-generating loader that is based on the code generator of the MacOberon system [Fra90a, Fra90b, Fra91, BCF92, Fra93a, Fra93b]. It makes use of the idea of *code-copying*, as outlined in the previous chapter, for which a preprocessed syntax tree, as it results from semantic-dictionary encoding and in which source-level common subexpressions and common designators have been factorized, offers an ideal basis. The strategy of initializing the semantic dictionary with entries based on the objects occurring in the symbol table is able to further enhance the efficiency of code generation.

Consider an object of the *variable* category in the symbol table. No further discrimination is usually made in the compiler front-end between different kinds of variables; for example, with respect to whether they are global to a module or local to a procedure. This distinction is left to the code generator, which for each variable reference has to determine what particular kind of variable it is handling, i.e., which addressing mode is to be employed.

Within an SDE-file, however, all references to variables are based on entries that are placed into the dictionary in an initialization step. The important insight is that the *meanings* of the individual entries in the decoder's dictionary need not directly match those of the corresponding entries used in the encoding step, as illustrated in Figure 3.6. Hence, it is possible to simplify the task of code generation by performing a certain amount of **preprocessing** at the time of dictionary initialization, discriminating more finely between different variants of symbol-table objects on the decoder's side than is necessary in the encoder.

```
VAR glob: INTEGER;

PROCEDURE P(VAR vp: INTEGER);
    VAR loc: INTEGER;
BEGIN
    ...
```

Source Program

Index	Class	Info
i	variable	glob
...	...	...
j	variable	vp
...	...	...
k	variable	loc

Semantic Dictionary
Used for Encoding

Index	Class	Info
i	**abs-adr-var**	adr(glob)
...	...	...
j	**mem-ind-var**	off(adr(vp))
...	...	...
k	**fp-rel-var**	off(loc)

Semantic Dictionary
Used for Decoding

Figure 3.6: Different Semantic Dictionaries for Encoding and Decoding.

The implemented code-generating loader for the MC68020 [Mot87] performs some of this preprocessing. For example, it differentiates between global variables (absolute addressing), direct local variables (frame-pointer relative addressing) and indirect local variables (memory indirect addressing, used for variable parameters). Consequently, three different kinds of dictionary entry are used to represent these different kinds of variables, and all information required for address generation is gathered in the semantic dictionary already when it is initialized. Since in typical programs most variables are referenced more than once, this strategy usually accelerates the code-generation process.

Decoding SDE-Files

After all the previous explanations, the design of the *SDE-decoding code-generating loader* then becomes quite straightforward. First, the global symbol table is read from the SDE-file. Next, the semantic dictionary is initialized as discussed before. Then, dictionary indices are read in succession and processed as is outlined below:

```
< read an index idx and look up the corresponding entry E := dict[idx] >

IF   E.copy-safe   THEN
    < duplicate the code between E.beg and E.end at the current PC location >
ELSE
    IF   < E is a template >   THEN
        < input undefined descendants of E recursively >
        < possibly create further templates heuristically >
        < create a new complete entry E from template and descendants >
    END;

    E.beg := PC;
    < generate code for E, updating the E.copy-safe flag appropriately >
    E.end := PC
END;
```

The **copy-safe** property needs to be kept track of for every complete dictionary entry. It indicates whether code for the corresponding expression can be produced simply by duplicating an instruction sequence generated earlier. Templates are never copy-safe.

A dictionary entry is copy-safe if there are *no pending fix-ups* within the resulting code fragment, and if the code is *position-independent*. In the implemented code-generating loader for the Motorola 68020, these criteria are fulfilled by all full entries with the exception of those describing expressions that contain calls of local functions. These calls are translated into relative branches, which have a different branch distance at every occurrence.

In fact, code-copying was the main reason why it was decided not to translate calls to external procedures into relative branches, which would have been easy since all addresses in external modules are known when generating code on-the-fly. This was the single instance in which speed of execution was traded for speed of code generation. However, the performance of the two call instructions does not differ by much, while external calls occur often in typical modules so that one can benefit from code-copying frequently.

Miscellaneous Implementation Details

There is a multitude of smaller design decisions in any project such as the one described here. Many of these decisions may be rather ad-hoc, while the motivations behind others may long be lost before they can be documented. The following list is by no means complete, but illustrates some of the finer points that are not mentioned elsewhere in this thesis.

- *Symbol Table.* From what has been said before, the reader might have obtained the impression that the complete symbol table is stored contiguously at the beginning of the SDE-file. This is not entirely correct. In fact, only the global scope is stored at the front of the SDE-file, while the remaining scopes follow the dictionary indices that represent the corresponding procedure entries. This provides for a more efficient storage management in the decoder, allowing a *stack mechanism* to be used not only for the dictionary, but also for the symbol table.

- *Constants.* In the currently implemented system, unnamed literal constants are actually not part of the symbol table, but inserted literally into the stream of dictionary indices, following the index of a "constant" template that implicitly indicates the type of the constant. For each constant represented in this way, a complete entry is constructed and added to the global end of the dictionary, so that further occurrences of the same constant can be referenced directly by an index. The constant *NIL* corresponds to a reserved word of the Oberon language and is placed into the dictionary during its initialization, as are the pre-defined constants *TRUE* and *FALSE*.

- *Storage Allocation Primitives.* In Oberon, calls to the standard procedures *NEW* and *SYSTEM.NEW* are usually translated into dedicated supervisor calls, so that only the appropriate trap vector number need be statically known to the compiler, but not the routine that implements it. In the absence of memory protection on the target machine, it makes no sense to utilize the expensive supervisor call mechanism when the actual address of the Kernel routine that implements the supervisor instruction is obvious at the time of code generation. Hence, the current implementation maps these two storage-allocation primitives onto ordinary procedure calls, which not only shortens the instruction sequence, but also accelerates the call.

4. Modular Design of the Implementation

The previous chapters have introduced the technique of *semantic-dictionary encoding* and have presented an *Oberon-to-SDE compiler* and a *code-generating loader*. Up to this point, however, it has been disregarded completely that the implemented system has a **modular structure**. This chapter gives an overview of this modular architecture and points out some alternatives to the chosen approach.

Modular Architecture

The implemented system is based on the *Oberon System* [WG89, WG92], specifically its *MacOberon* implementation [Fra90a, Fra90b, Fra93a] for *Apple Macintosh* computers [App85]. The Oberon System supports **separate compilation** with static interface checking of programs written in the programming language Oberon [Wir88]. It also offers **dynamic loading** of individual modules, meaning that separately compiled modules can be linked into an executing computing session at any time, provided that their interfaces are consistent with the interfaces of the modules that have been loaded already. In MacOberon, dynamic loading is accomplished by a **linking loader**, which modifies the code image retrieved from an object file in such a way that all references to other modules are replaced by absolute addresses.

The new implementation adds a *second loader* to MacOberon, namely a **code-generating loader** that processes files containing semantic-dictionary-encoded programs (SDE-files). It has been possible to integrate this code-generating loader transparently into the existing MacOberon environment, in the sense that SDE-files and native object-files are completely interchangeable and can, therefore, import from each other arbitrarily. Depending on a *tag* in the file (first two bytes), either the native linking loader or the code-generating loader is used to set up the module in memory and prepare it for execution.

The code-generating loader has not been incorporated into the core of MacOberon, but constitutes a self-contained module package at the application level. The existing module loader was modified slightly, introducing a *procedure variable* into which an **alternate load procedure** can be installed. Once initialized, this procedure variable is up-called whenever an abnormal tag is detected in an object file, and so initiates the passage of control, from the linking loader that is part of the system core, to the code-generating loader external to it. The installation of such an alternate load procedure is considered a privileged operation, analogous to the modification of an interrupt vector.

Besides the obvious advantages during the design and testing phases, this architecture permits us to view machine-independence quite naturally as an *enhancement of an existing system*. It also hints at the possibility of adding successive external code-generating loaders as time progresses and standards for machine-independent object-file formats emerge. Several such formats could in fact be supported simultaneously, among which one would distinguish by different file-tags.

Discussion: On System Architecture

In the implemented system, the code-generating loader is installed by an explicit call to *Modules.This*. However, it would be entirely possible to issue such a call during the initialization of module *Modules* itself, leading to a three-stage bootstrap of successively more powerful loading capabilities:

1. Boot Loader
2. Native Linking Loader
3. Code-Generating Loader

The only task of the boot loader is to install the native linking loader and the core operating system routines required by it, such as file-system-access capabilities. The native linking loader in turn sets up the code-generating loader from a pre-compiled object file. All higher modules, even such essential ones as the display system, may then be represented by SDE-files, as long as the algorithms contained in them can be formulated machine-independently.

One immediately senses the possible implications of this. Entirely feasible, and implemented in an experimental system variant, is a mode of operation in which external inputs during the bootstrapping process modify the default behaviour of the system. For example, depressing the key labelled "x" on the keyboard during startup could turn on *array subscript checking* for all library modules being loaded during this phase, while it would normally be turned off. Likewise, symbolic-debugging information could be generated only when needed, and need not take up memory space in configurations that are used solely for running finished applications.

It is also notable that, after an initial bootstrap, one could have done away with machine-specific object-files altogether by including a code-generating loader in the boot-file and providing a mechanism for generating new boot-files. In fact, with some simple modifications for supporting the generation of *relocation information*, the code-generating loader itself could be employed for the generation of new boot-files. The contents of a boot-file are very similar to those of the module area in the system heap at run-time, except that the former contains also a list of references that need to be relocated.

A similar approach has been taken in the Smalltalk-80 system [Gol84], in which there are only target-machine-independent source-files and an *image file* containing the **snapshot** of a compiled system state. The image file needs to contain an interpreter or compiler, so that the portable source-files can be used at all. After modifying the run-time environment, a new image file can be generated that incorporates all of the changes applied to the previous image.

Unfortunately, however, systems in which all machine-dependent parts are encapsulated within a boot-file have a drawback, which is that a change in any of the modules contained in the boot-file requires the generation of a whole new boot-file. It was exactly this inflexibility that modular architectures were trying to avoid in the first place. For this reason, the idea has not been pursued any further for the time being.

Execution Frequency Hierarchy Considerations

Programs are usually *compiled* far less frequently than they are *executed*. Therefore, it is worthwhile to invest some effort into compilation, because the benefit will be repeated. The same holds for program *loading* versus *execution*. While a program is normally loaded more often than it is compiled, the individual machine instructions are executed even more often, many times on average per load operation. Unfortunately, we need to shift workload unfavourably when we employ a loader that performs code generation. This can be justified only if important advantages are gained in return, or if the resulting code quality is increased. After all, it is execution speed that matters ultimately.

From the outset, three desired properties were therefore put forward that a system incorporating a code-generating loader should possess in comparison to a system employing a traditional compiler and linking loader:

1. run-time performance should be at least as good
2. loading time should not be much worse
3. source-encoding time should be tolerable

These requirements could be met. Benchmarks in the next chapter will show that it was not only possible to construct a fast code-generating loader for *Motorola 68020* processors [Mot87], but that the native code produced by it is of high quality. Indeed, although its method of code-generation is quite straightforward, the code-generating loader is sometimes able to yield object code that would require a much more sophisticated code-generation strategy in a regular compiler. This is mainly due to the fact that the absolute addresses of all imported objects are known at loading time, which enables the code-generating loader to optimize access to them; for example, by using short displacements instead of long ones.

The time required by the code-generating loader for creating native code on-the-fly turned out to be quite acceptable, too. Loading with code-generation takes only slightly longer than loading of a native object-file with linking; the difference is barely noticeable in practice. A partial reason for this is that the Oberon system has a modular structure, in which many functions are shared between different application packages. In traditional systems, these common functions would be replicated in many applications and statically linked to each of these applications.

As a consequence of Oberon's modular design, each new application adds just little code to an already running system. Hence, at most times during normal operation, the code-generating loader need only process the moderate number of modules that are unique to an application, while other modules that are also required will already be in memory from past activations of other applications.

Interchangeability of Object Files

SDE-files and their native-object-file counterparts are completely interchangeable in the implemented system. This flexibility doesn't come as readily as it may seem. Not only does it require that the code-generating loader is able to interpret *native symbol-files* and the corresponding entry-tables in memory, but also that information can be communicated from the new load architecture to the old one, in formats that are *backward-compatible* with the native compiler and loader.

In the present case, the more difficult part of this problem had an almost trivial solution, since the two varieties of "object" files in the implemented system share a *common representation of symbol-table information*. As documented in a paper by the author [Fra93b], a **portable symbol-file format** had been introduced into MacOberon some time ago, leading to improvements in performance completely unrelated to portability. But now, there were further benefits from the previous investment into machine-independence:

SDE-files use exactly the same symbol-table encoding as do the symbol files of the native compiler. Furthermore, the symbol table is embedded within each SDE-file in such a way that its front-most part would constitute also a valid symbol file (Figure 4.1). Consequently, the native MacOberon compiler cannot distinguish SDE-files from its regular symbol-files. It is able to compile modules that import libraries stored in SDE form, as it can extract the required interface information directly from SDE-files.

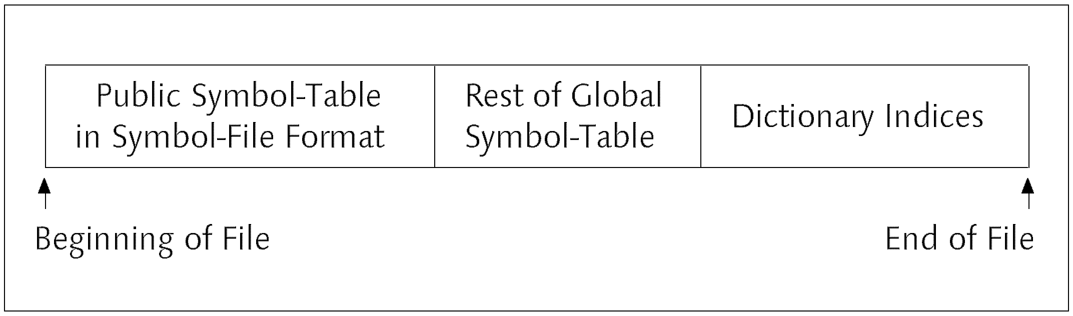

Public Symbol-Table in Symbol-File Format
Rest of Global Symbol-Table
Dictionary Indices
Beginning of File
End of File

Figure 4.1: Embedding of Symbol File within SDE-File

Apart from the native compiler, the *native loader* need also be able to handle modules that import machine-independent libraries. This is achieved by enabling the code-generating loader to construct on-the-fly not only object code, but also all of the remaining data structures that are usually part of a native object-file, such as *entry tables*. Once that a module has been loaded from an SDE-file, it loses all aspects of machine-independence, and can henceforth be maintained (enumerated, unloaded, etc.) by the regular MacOberon module manager.

Discussion: On Interfacing Without a Common Symbol File Format

Without a common symbol-table encoding, the task of interconnecting machine-independent modules in the SDE format with others in a regular object-file format would have been more difficult. Indeed, since SDE-files are independent of the target architecture, we might want to use them on more than one kind of machine, on which many different native symbol-file formats could be in concurrent use. In the following discussion, the adjective *native* refers to the formats used on the eventual target machine, while *portable* denotes the machine-independent SDE format.

Enabling *native clients* to import *portable libraries* is quite straightforward, although it may not be so simple to implement. It requires that each local compiler on every target machine can understand the portable symbol-table encoding in addition to its own, and that each code-generating loader can provide entry tables matching this symbol information, in the format required by the corresponding native loader. For some target machines, fulfilling the first requirement may entail a significant amount of programming, because the portable symbol-table encoding contains no machine-dependent information such as type-sizes and offsets. Consequently, local compilers need to be able to calculate these values on-the-fly when reading the symbol table from an SDE-file.

The reverse import relationship, however, is even more problematic. Consider a portable module that imports module *Files*. Implementations of *Files* are inherently non-portable, so that they need to be compiled by a native compiler always. Nevertheless, a portable client module should be linkable to the local *Files* module of an arbitrary target machine, assuming that the different *Files* modules all have the same interface. But how do we *ensure* that all of these interfaces on different machines are indeed compatible with the portable client, and how do we encode references to them in the portable object-file?

A possible solution to this problem, which has actually been implemented in a precursor version of the present system, starts by defining a second, portable interface for each of the native library modules in question. These portable interfaces serve as **place-holders** for a portable subset of the actual interfaces found on different machines. They define capabilities that are expected to be present on every machine, so that native interfaces are in fact allowed to be a superset of their portable counterparts. Portable symbol-files are then generated from these "place-holder" interfaces, and used whenever portable clients are compiled.

For each target machine, one then constructs a tool called a *localizer*. The localizer connects each portable "place-holder" interface with the corresponding native one, reading both the native and the portable symbol file. If the native version of the module includes every feature specified in the portable interface, the localizer creates a *mapping file* that will tell the code-generating loader which local entry number is associated with which feature in the portable interface. This mapping mechanism is illustrated in Figure 4.2. Each mapping file contains also the keys of both related symbol files, so that the validity of a mapping can be verified by comparing module keys.

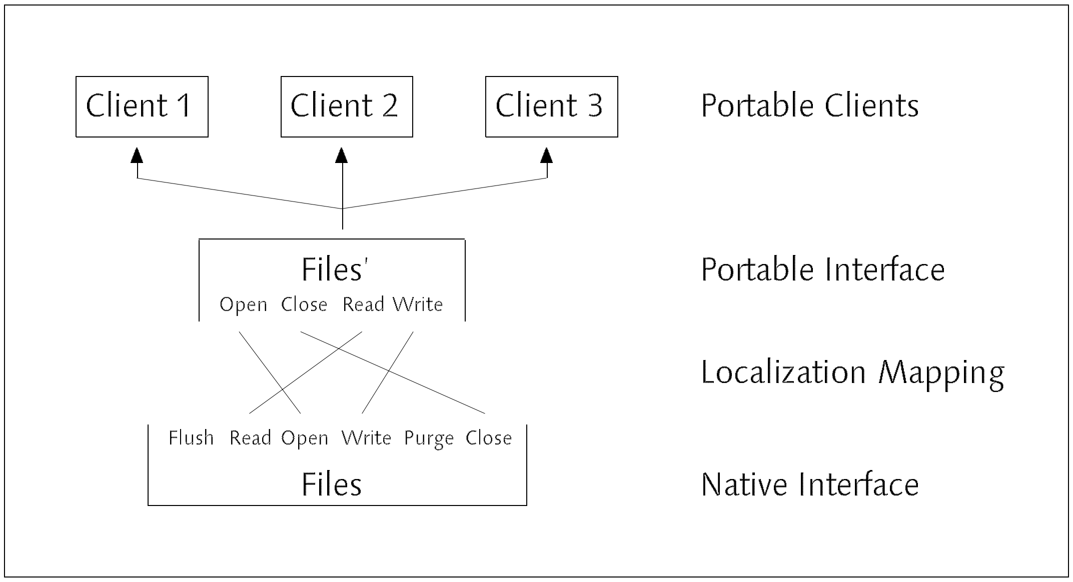


Figure 4.2: Localization of an Interface

Levels of Symbolic Information

As mentioned in [Fra93b], machine-independence of symbol files mandates that the size of each data type (as well as offsets and other machine-dependent data) is calculated only at the time that the symbol file is read, because it may differ from machine to machine. This has consequences when parts of a data structure can be hidden, as is possible in Oberon. Since it is impossible to calculate the total size of the invisible components beforehand, portable symbol-files need to contain a *full structural description* (the *names* of the hidden parts are obviously not needed) of every exported feature, including all non-exported parts.

This has the undesirable effect that a portable symbol-file (and thereby its key) may change when certain modifications are made in the *hidden part* of the corresponding module, at least as long as a conventional combination of compiler, linker, and loader is used. When employing a code-generating loader instead, this situation changes. Upon closer analysis of a symbol file's information content, we find that ordinary compilers require more information than code-generating loaders:

– Code-generating loaders need only know about *exported names and structure*. They never require any addresses, offsets, or sizes, nor do they need information about invisible parts of an exported object. This is because all inter-module references are handled strictly symbolically and are not resolved until the loading phase, which includes code generation.

– A compiler generating *native code* requires the same symbolic information, plus knowledge about all *invisible fields and structures* to which references exist from exported types. This is because the native compiler has to calculate *offsets*, on which the addressing modes of some instructions may depend; certain offsets may also appear literally in the object code.

It would therefore in principle be possible to use a two-tier structure for storing the symbol table, in which the second layer reveals some finer grained details of the implementation. This mechanism has not been implemented because it adds complexity while applying only to a side-issue of the dissertation project, but the following elaboration on the idea of two-tier symbol-files suggests a possible evolution of the current work. Consider the following structure of a symbol file:

```
Interface Part
   re-exported modules
   exported constants
   exported types (not mentioning invisible fields)
   exported global variables
   exported procedures
```

Offset Calculation Part
  more imported modules (containing the types of invisible fields)
  invisible types of invisible fields
  invisible fields (of types mentioned in Interface Part)

With this organization, all client modules that are represented by SDE-files are impervious to changes in the hidden part of a library module, because they depended only on the *interface part* of the library's symbol file. Whenever a module is compiled under such a scheme, both parts of the symbol file are updated, regardless of the type of "object" file generated. However, changes in the *offset calculation part* can affect only those clients that have been compiled into directly executable object code.

In systems supporting two-level symbol-files, it is therefore advantageous to use as few native object-files as possible, in order to reduce intermodule dependencies. Moreover, the penalty for using native object-files is greater at a higher level in the hierarchy than at a low level, because interface invalidations propagate upwards. It has always been good software engineering practice to isolate machine dependencies in low-level modules and strive for portability higher up in the module hierarchy. Having a technical argument might further increase the acceptance of these rules.

The interface part of the symbol file may even be changed to include **per-feature keys** instead of a single key for the whole interface. This would further reduce the interdependency of the portable modules in the system, as it would make possible a **link-by-name** facility. A client module can be connected to a library if all of the requirements of the client can be fulfilled by features of the library. Instead of a crude comparison of whole interfaces, we could therefore compare *exported objects* on one side with *imported objects* on the other. Changes in a module's interface would then only invalidate those portable clients that made direct use of the features that were modified, but left all other portable clients unperturbed. By appending a separate version key to every name, we would be able to use a simple string comparison to determine whether the requirements of one module conform to the features of another, without having to repeat the compiler's structural analysis.

Note that some key changes could be avoided also if the native compiler were allowed to change the ordering of fields in record types (i.e., moving the visible fields to the lowermost offsets). However, the native compiler requires also the *size* of these types. Adding or removing data fields therefore necessarily invalidates clients represented by native object-files.

5. Benchmark Results

This chapter provides some data on the performance of the implemented system and discusses these results. Four different Oberon application-packages have been used in the following benchmarks. *Filler* is a program that draws the Hilbert and Sierpinski varieties of space-filling curves onto the screen. *Hex* is a byte-level file editor of medium sophistication. *Draw* is an extensible object-oriented graphics editor [WG92], of which the three main modules and three extension modules are included in the survey. *Edit* (formerly called *Write*) is a relatively sophisticated extensible document processing system [Szy92], five modules of which are studied here.

Memory and File-Store Requirements

Figure 5.1 gives an impression of the compactness of SDE-files, the influence of the presence of run-time integrity checks on code size, and the memory requirements of the code-generating loader. Its first three columns compare the sizes (in bytes) of native MacOberon object-files with those of SDE-files. Two different values are given for the former, reflecting different levels of run-time integrity checking. SDE-files contain enough information to generate code with full integrity checking, but the user need only decide at loading time whether or not he requires code that includes these run-time checks, and which ones should be included.

The column labelled *"Native –"* in the table below displays the size of native object-files that include reference information for the MacOberon post-mortem debugger, but no extra code for nil-checking, index-checking, type-checking, nor for the initialization of local pointers to NIL. Conversely, the column labelled *"Native +"* gives the sizes of native object-files that include not only reference information, but also code for the aforementioned run-time checks. Neither kind of object file includes symbolic information for interactive debugging, which can be added by enabling a further option.

The fourth column of Figure 5.1 shows the maximum size to which the semantic dictionary grows during compilation and code-generation. The code-generating loader requires about 80 times this number of bytes as temporary storage while loading a module.

Module	Native –	Native +	SDE	Dictionary
Filler	2900	3236	1232	1003
Hex	10810	12678	4480	1308
Graphics	12692	15935	5281	1382
GraphicFrames	10045	12186	4273	1549
Draw	6033	7133	2193	1498
Rectangles	3255	4297	1378	1271
Curves	5810	7340	2168	1287
Splines	4611	5659	1955	1566
TextFrames	30687	37491	11667	1892
TextPrinter	11503	12537	4494	1486
ParcElems	15259	17540	5273	1437
Edit	11670	13431	5157	1659
EditTools	18756	20898	7397	1488
	144031	170361	56948	

Figure 5.1: Object-File Size (Bytes) and Dictionary Size (Entries)

This data shows that on average, SDE-files are about 2.5 times more compact than native MacOberon object-files not containing run-time integrity checks, and about 3 times more compact than native files incorporating these checks. It is also noteworthy that the maximum size, to which the semantic dictionary grows during encoding and decoding, is not proportional to the overall size of a module, but only roughly proportional to the length of the longest procedure in a module. This seems to level out at a relatively small value in typical modules, much smaller than anticipated originally.

Figure 5.2 indicates how much information is output into memory by the code-generating loader. The first column repeats the sizes of SDE-files from the previous table. The second and third columns give two different values for the size of the object code generated on-the-fly, depending on whether or not run-time integrity checks (in the same combinations as before) are emitted. The remaining columns list the sizes of dynamically-generated constant data (including type descriptors), reference data for the Oberon post-mortem debugger, and link data. The latter comprises entry tables generated on-the-fly, so that native client modules can later be connected to dynamically generated library modules.

Module	File	Code –	Code +	Const	Ref	Link
Filler	1232	2352	2682	139	244	8
Hex	4480	9224	11020	267	1094	12
Graphics	5281	9036	12140	799	1452	204
GraphicFrames	4273	8268	10354	433	824	76
Draw	2193	4754	5752	236	451	60
Rectangles	1378	2644	3596	104	292	20
Curves	2168	5042	6462	100	439	24
Splines	1955	3812	4806	137	437	20
TextFrames	11667	25878	32536	629	2843	156
TextPrinter	4494	9176	10180	478	1367	40
ParcElems	5273	12944	15180	493	1085	52
Edit	5157	9368	10940	555	1098	72
EditTools	7397	14844	16886	723	2009	116
	56948	117342	142534	5093	13635	860

Figure 5.2: Sizes of SDE-Files and of Dynamically-Generated Data (Bytes)

It is notable that SDE-files encode programs more than twice as densely as object code for the MC68020 architecture [Mot87]. If code that includes run-time checking is considered instead, the factor becomes even 2.5. This is in spite of the fact that SDE-files can additionally also serve as symbol files and contain reference information as well.

Performance

Unless stated otherwise, the following benchmarks were all carried out under *MacOberon Version 4.03* on a *Macintosh Quadra 840AV* computer (*MC68040* Processor running at 40MHz, equipped with a graphics accelerator card), using version *7.1.1* of the Macintosh System Software. All results are given in real time, i.e. the actual delay that a user experiences sitting in front of a computer, with a resolution of 1/60th of a second. Each benchmark was executed after a cold start, so that no files were left in the operating system's file cache between a compilation and subsequent loading, and the best of three measurements is given in each case. Every attempt has been made to present the system as a regular user would experience it in everyday use.

Figure 5.3 presents the times (in milliseconds) required for compilation and for module loading. The first two columns compare the native compilation times of the MacOberon compiler with the times that the Oberon-to-SDE compiler requires for generating an SDE-file out of an Oberon source program. The remaining two columns report the loading times for both varieties of object file, including disk access, and, in the case of SDE-files, also including on-the-fly generation of native code. On average, 10% of the object code emitted by the code-generating loader can be generated by code-copying. All timings apply to the situation in which no run-time integrity checking is used.

Module	Compile	Encode	Ld Native	Ld SDE
Filler	583	633	83	100
Hex	666	900	100	116
Graphics	766	1083	116	166
GraphicFrames	650	950	66	116
Draw	516	683	66	100
Rectangles	400	500	50	83
Curves	450	650	50	116
Splines	433	583	66	83
TextFrames	1416	2183	216	316
TextPrinter	666	950	100	133
ParcElems	750	1150	116	200
Edit	716	1166	100	166
EditTools	900	1483	150	233
	8912	12914	1279	1928

Figure 5.3: Compilation versus SDE-Encoding and Module Loading Times (ms)

As can be seen from these timings, Oberon-to-SDE encoding on average takes about 1.5 times as long as normal compilation. This factor could certainly be further reduced by using more complicated sorting structures for the semantic dictionary (see Chapter 3). However, more important is the speed of loading with code generation. This currently takes about 1.5 times as long as normal loading, but the times spent directly on module loading tell only half the story.

Figure 5.4 presents a more realistic measure of loading time, as it takes into account not only the time required for loading an application, but also the duration of *loading and displaying a typical document*. The following timing values indicate how long (in milliseconds) a user must wait after activating a document-opening command before he can execute the next operation. Commands take longer the first time that they are issued, because the modules of the corresponding application have to be loaded as well. Subsequent activations then take up much less time.

Command	Native –	SDE –	Difference
first Draw.Open Counters.Graph	1333	1450	+ 9%
subsequent activations	700	700	
first Edit.Open OberonReport.Text	1400	1766	+ 26 %
subsequent activations	883	883	

Figure 5.4: Command Execution Times with Checks Disabled (ms)

Figure 5.5 demonstrates the effect of run-time integrity-checking on loading time. The measurements of Figure 5.4 have been repeated, but with native object-files that incorporate run-time integrity checks and are therefore larger, and with on-the-fly emission of the same checks in the code-generating-loader.

Command	Native +	SDE +	Difference
first Draw.Open Counters.Graph	1366	1466	+ 7%
subsequent activations	700	700	
first Edit.Open OberonReport.Text	1550	1800	+ 16%
subsequent activations	916	916	

Figure 5.5: Command Execution Times with Checks Enabled (ms)

The timings in Figures 5.4 and 5.5 show that, in practice, on-the-fly code-generation is already almost competitive to dynamic loading of pre-compiled native code from regular object-files. This applies to current state-of-the-art CISC hardware, and is in part due to the fact that disk reading is relatively slow. The compactness of the SDE representation speeds up the disk-access component of program loading considerably, and the time gained thereby counterbalances most of the additional processing necessary for on-the-fly code generation. This argument is supported by a comparison of Figures 5.4 and 5.5. It seems to be more efficient to generate run-time checks on the fly than to inflate the size of object-files by including them there.

A noteworthy trend in hardware technology today is that processor power is rising more rapidly than disk access times and transfer rates are falling. This trend is likely to continue in the future, which means that hardware technology is evolving in favor of the ideas proposed in this thesis. Consider Figures 5.6 and 5.7, which repeat the timings of Figure 5.5 for different processors of the MC680x0 family, using the identical external disk drive unit for all experiments.

Machine	Native +	SDE +	Difference
Macintosh II (16MHz MC68020)	5683	7966	+ 40%
Macintosh IIx (16MHz MC68030)	4300	5933	+ 38%
Macintosh IIfx (40MHz MC68030)	1833	2283	+ 25%
Quadra 840AV (40MHz MC68040)	1366	1466	+ 7%

Figure 5.6: Time for Draw.Open Counters.Graph *on Different Machines (ms)*

Machine	Native +	SDE +	Difference
Macintosh II (16MHz MC68020)	4733	8850	+ 87%
Macintosh IIx (16MHz MC68030)	3600	6750	+ 88%
Macintosh IIfx (40MHz MC68030)	1550	2400	+ 55%
Quadra 840AV (40MHz MC68040)	1550	1800	+ 16%

Figure 5.7: Time for Edit.Open OberonReport.Text *on Different Machines (ms)*

Extrapolating from these results, there is reason to believe that on-the-fly code-generation from small SDE-files may eventually become faster than dynamic loading of larger native object-files, unless of course secondary storage universally migrates to a much faster technology. One way or the other, on-the-fly code-generation will definitely become faster in absolute terms as clock speeds increase further, so that the relative speed in comparison to native loading should not much longer be of importance anyway. Ultimately, the speed of loading needs to be tolerable for an interactive user – that is all that matters.

Code Quality

The last point that needs to be addressed concerns the quality of code that is generated dynamically. Figure 5.8, by way of a popular benchmark, compares the quality of code generated on-the-fly with that generated by the *Apple MPW C compiler for the Macintosh (Version 3.2.4)* with all possible optimizations for speed enabled (*"-m -mc68020 -mc68881 -opt full -opt speed"*). The generation of integrity checks was disabled in the code-generating loader, because no equivalent concept is present in *C*.

Benchmark	SDE –	MPW C
Permutation	**83**	113
Towers of Hanoi	**83**	121
Eight Queens	50	**43**
Integer Matrix Multiplication	**150**	173
Real Matrix Multiplication	**133**	171
Puzzle	**800**	**800**
Quicksort	66	**61**
Bubblesort	117	**88**
Treesort	**83**	> 1000
Fast Fourier Transform	133	**123**

Figure 5.8: Benchmark Execution Times (ms)

The table lists execution times in milliseconds (less is better). Due to processor cache effects, these timings can vary by as much as 15% when executed repeatedly; the figures give the best of three executions. The *C* version of the *Treesort* benchmark exceeded all meaningful time bounds, due to apparent limitations of the standard Macintosh operating system storage-allocator. The code-generating loader is not bound by these limitations, as it generates calls to the storage-allocator of *MacOberon*, which includes an independent memory-management subsystem.

From this data, it can be inferred that the code-generating loader emits native code of high quality that can compete with optimizing *C* compilers. In some cases, it surpasses even the output of the official optimizing compiler recommended by the manufacturer of the target machine, which is orders of magnitude slower in compilation. It should be possible to improve the remaining cases in which the code-generating loader is currently inferior, without sacrificing much of the speed of on-the-fly code generation. Since the code-generation efficiency of the code-generating loader is rooted in the compactness of the abstract program representation, rather than in any machine-specific details, it should also be possible to duplicate it for other architectures.

The *Apple MPW C* compiler requires about 6.9 seconds for compiling a C source program that performs the series of benchmarks listed above, and a similar time additionally for linking, compared to 1.1 seconds that are needed for encoding a corresponding Oberon program into the SDE file-representation and 233 milliseconds for loading it with on-the-fly code generation (including file access after a cold start).

6. Portability and Software Components

It has already been mentioned that semantic-dictionary-encoded programs are *independent of the eventual target machine*, which means that they are, in principle, *portable* between different machine architectures. This chapter examines the notion of **software portability** more closely; it presents some existing definitions and approaches to portability and introduces a new, concise and practical definition of **upward-compatibility**. It is then argued that fast on-the-fly code-generation from a machine-independent intermediate representation constitutes an *enabling technology for a software-component industry*, because it would simplify the distribution and maintenance of such components considerably, without resulting in a loss of code efficiency or user-convenience.

Portability

Portability is one of the more elusive concepts in computer science. There is in fact only one aspect of portability about which there seems to exist a general consensus, namely that it is beneficial. Probing further, we soon find that there are many diverse opinions of what portability actually amounts to in practice. This is somewhat surprising, considering the economic importance of the concept and the widespread use of the term "portable", which seems to be quite clear intuitively.

Most computer-literate people would probably agree that portability somehow relates to "usability of the same software on different machines". A casual remark by Dahl [Dah84] describes portability as "having the same meaning for software as **compatibility** for hardware". Again, most of us would probably consent to that statement, but this does not get us much further. Unfortunately, the term "compatible" is equally ambiguous as "portable".

Part of the problem of the portability debate is that so many diverse things have been mixed up into it. Moreover, the emphasis has shifted significantly over time. Not long ago, a major thread of the portability argument concerned the **physical portability** of programs and data, specifically the problems of different character sets (e.g., *ASCII* versus *EBCDIC*) and the multitude of storage media (e.g., *7-track* versus *9-track* magnetic tape). In 1975, Waite [Wai75] reported that "getting it into the computer" was often more difficult than "getting it to run". Those were the days when the first design decision at the beginning of a programming project concerned the choice of a character set.

Today, physical portability of *programs* is no longer of any concern and physical portability of *data* is becoming less of an issue because rising processor speed makes it possible to re-code data transparently on-the-fly. The problems of media compatibility have been moderated by evolving standards and by dropping hardware costs, which have made it possible to support a multitude of formats concurrently. On top of all this, pervasive networking is successively eliminating the need to use physical media at all for transporting machine-readable information.

Other no longer relevant issues relate to former limitations in compilers and operating systems, for example, regarding the length of identifiers in source programs – it is becoming hard to appreciate that this was ever a problem. Tanenbaum et al. [TKB78] enumerate exhaustively the areas in which programs were most unlikely to be portable in 1978. Half of their concerns simply no longer apply in 1993. Unfortunately, however, new challenges have emerged in the meantime, such as the mastery of parallelism in multiprocessor systems. These will keep us busy for a while.

Portability of Programs

The cost of developing software is rising steadily as programs get ever more complex in order to use the additional capabilities of emerging hardware to the fullest. At the same time, hardware costs are falling, making the disproportionately high price that we pay for software more apparent. An obvious solution for keeping in check the exploding costs of software is to spread the cost of development by selling it in large volumes. However, in view of the many different computer architectures that coexist, such mass-market software must necessarily be portable so that it is not confined to the market segment of just one specific type of target machine.

A necessary precondition for portable software is the use of a **high-level language** in its construction, because all alternatives would render it machine-dependent. In this context, a high-level language is one that *abstracts from the underlying hardware*. However, the obstacles to portability that remain even when high-level languages are used are often underestimated, as relatively few computational processes do not depend upon their environment in some way.

Nevertheless, most algorithms can be parametrized in a manner that facilitates easy adaptation to a particular target environment, although this usually requires some planning ahead ("design for portability"). A common solution for handling machine dependencies in a program is to isolate into a single **module** those parts of the program that are machine-dependent. Hopefully, only this machine-dependent module needs to be modified when the program is subsequently ported to another machine.

Some authors make a finer distinction between programs that are usable on several machines without any changes whatsoever, and those that require adjustments. Hague and Ford [HF76] use the term **portable** in a restrictive way to describe only those programs that can be compiled and will then execute correctly on other machines completely unmodified. If the necessary changes for such a move are capable of mechanical implementation via a preprocessor, they call the software **transportable**.

Others use a less rigid definition of the term portable. A widespread practice is to call a program portable if the effort of moving it to a new environment is much less than the effort of rewriting it for the new environment [BH76, Fox76]. For some authors, a program may even be called portable if it is so well *documented* that it can be rewritten completely for another machine with the help of the original design documents [Wal82]. Dahlstrand [Dah84] interprets portability as a *measurable quantity* and suggests to express it as "the percentage of lines that could be left unchanged when porting an application".

Portability of Numerical Software

Numerical software is written almost exclusively in *FORTRAN*, so that the "portability" effort in this context concerns itself only with the incompatibilities that arise out of the use of various language dialects and the particularities of different target environments. Tools such as the *PFort* verifier [Ryd74], which checks a *FORTRAN* program for adherence to a portable subset of the language, are useful in detecting these potential impediments to portability.

Several existing numerical software libraries have been designed to be portable. The central idea of a portable library is to maintain only a single **master version** of each routine, and derive all variants from it mechanically [Boy76]. Maintaining just one version of a program offers substantial economic advantages.

The simplest approach to such a master-file solution is a **markup scheme**, under which a programmer inserts annotations into a *FORTRAN* source file, which specify the changes that need to be made to produce other versions. These annotations (which are specially marked comments) are interpreted by a **preprocessor**, which outputs a version of the *FORTRAN* routine that is tailored to a particular target environment. Examples of markup schemes are the *Specializer* [Kro76] and the *IMSL Converter* [Air76]. The latter includes also a special "scan" option that can perform a precision conversion without requiring specially marked comments to trigger its operation. It is able to change type declarations, constants, and function names automatically as required.

In the *Master Library File System (MLFS)* [HF76, RH77] of the *Oxford University Numerical Algorithms Group*, the control statements that differentiate between the different variants of a program are introduced automatically rather than by a human programmer. This is accomplished by the use of an **anti-editor**. The anti-editor is a tool that compares a particular executable version of a certain routine with the master version of the same routine. It then modifies the master version so that the executable variant can be generated from it automatically in the future.

An even more ambitious tool is used in the maintenance of the *LINPACK* library [DMB79]. *LINPACK* is designed to be completely machine independent and makes no use of machine dependent constants, input/output statements, character manipulation, the *COMMON* or *EQUIVALENCE* statements of the *FORTRAN* language, nor of mixed-mode arithmetic. It is therefore possible to automate completely the process of converting it for the use in different target environments. The *LINPACK* project uses a system originally called *NATS II* and later renamed to *Transformation-Assisted Multiple Program Realisation (TAMPR)* [BD74, Dri76]. In this system, programs are stored in a **canonical intermediate representation** that can be translated mechanically into variants of *FORTRAN* for several machines. The backward translation is also possible, so that existing programs originating on some particular machine can be added to the library. All knowledge about specific target environments in *TAMPR* is contained in the converter programs, and not in the individual master files. As a consequence, however, this system can handle only programs that have been designed for machine independence.

Unfortunately, numerical software is inherently portable only among machines that support the same **model of machine arithmetic**. For example, it is quite possible for an iterative algorithm not to converge when executed in a different precision than the one it was designed for. Likewise, the behaviour of different machines in the events of overflow and underflow may not be the same. The questions of numerical algorithm stability are intricate and leave many pitfalls to the mathematical layman attempting to adapt a scientific calculation from one machine to another. A partial answer has been provided in the form of a standard arithmetic model, namely the *IEEE Standard for Binary Floating-Point Arithmetic* [IEEE85].

Emulation, Abstraction Layers, and Intermediate Languages

The concept of portability is also closely related to that of **emulation**. Software that has been constructed to run in a certain environment *E* can be made to run in another environment *E′* by supplying an emulator within *E′* that provides the functions of *E*. Emulation is quite common and can be applied at different levels of abstraction. For example, most readers will be familiar with *terminal emulation programs* that simulate the functions of the classic visual display terminals (*HP 2645*, *IBM 3101*, *VT 100*, ...) on modern computers equipped with bit-mapped displays. Some more complex emulators allow software created for one processor to be run on another, by replicating the behaviour of the original target hardware on the other machine in software [BKM87]. There are emulators that provide the services of a whole operating system on a machine running a different operating system, by mapping the functions of the first onto equivalent calls to the second [Fra93a].

A closely related alternative to emulation is the introduction of a **common abstraction layer** into all target environments simultaneously. Software that is to be portable can then build upon this portable interface layer, in order to achieve independence of the target system. One could also say that the abstract system is emulated on every target system. Weiser et al. [WDH89] describe such an intermediate layer that abstracts from a concrete operating system and serves as a language-independent and operating-system-independent software system base.

The level at which a common abstraction is established may also be the target architecture for which the software is coded. A popular way of accomplishing software portability is by using a machine-independent, fairly low-level **intermediate language**, which can be easily mapped into a number of different assembly languages [Bro72]. Intermediate languages have already been discussed in Chapter 2.

Portability of Data

Processor architectures differ in the internal representations that are used for the various data types, with respect to the number of bits representing a certain value, and with respect to the specific meaning of the individual bits in such a representation. When information is transported from one computer system to another, a format conversion is therefore often necessary.

Data conversion requires that the source and destination formats are specified down to the level of the individual bit. For this purpose, Sibley and Taylor [ST73] propose a **data-definition language** that describes not only the logical data structures, but also how they are realized physically in a computer system. Atkinson [Atk77] goes one step further and reports of an actual implementation in which the conversion process has been automated. It uses an intermediate representation, described by a machine-independent *data language*, to transfer complex data structures across the boundaries imposed by differing machines, operating systems, and languages.

Unfortunately, however, systematic solutions such as these have never much caught on. Instead, most of the external data formats that are in use today are simply ad-hoc specifications that have become **de-facto standards** by virtue of their popularity. Quite often, they are based on the particular internal storage layout of the machine on which they originated. Whenever such values are input or output on a machine that has a different internal layout, they need to be converted. Luckily, the computations required for these conversions no longer carry much weight in comparison to the cost of the input and output operations. This is due to rising processor power relative to storage speeds.

There are even cases in which data portability is achieved by limited **hardware support** of data formats that are not considered native by the processor. A reason for this may be backward compatibility with older product lines. For example, in Digital Equipment's *Alpha* architecture [Dig92], some floating-point values are kept in memory in a representation that differs from the one that is used inside of processor registers. Alpha processors provide special load and store instructions for dealing with values in these "historical" formats, and include dedicated circuitry for re-arranging the bits of floating-point values accordingly as they pass between memory and processor.

Algorithmic Consequences

The differences in data representation between hardware architectures matter not only when values are transferred from one system to another. They have also direct consequences on the validity of portable programs. For computations involving real numbers, true portability is impossible unless the computation is parametrized in terms of certain models of machine arithmetic. This is the case because floating-point values are only *approximations* of real numbers, and the correspondence between actual values and machine numbers may vary for different machines.

On the other hand, **rational values** can be represented exactly in a digital computer (by quotients of integers). However, there are variations of representation, with respect to the number of bytes used, and with respect to byte-ordering. In order to guarantee universal portability, an algorithm must not depend on these parameters. Although byte-ordering is of no consequence for program portability in principle, it plays a role when data is to be *serialized*, e.g., represented externally or transferred over a network.

So what about a program in which a certain value N needs to be represented at some point of the computation? Obviously, there is a qualitative difference between machines on which N can be represented directly, and less capable machines that cannot process values of N's magnitude. On the latter type of machine, portability can be achieved only by emulation, which may be relatively costly.

Consequently, we need to differentiate between moving software to a "less capable" machine and moving it to a "more capable one". It would be wise to classify programs by a more refined notion than just "portable by recompilation", considering that portability by emulation may cause unreasonable expenses. In practice, programs are usable on a target machine only if no emulation is required for elementary computations [FL91].

I therefore propose a new concept that I call **upward-compatibility**, to replace the traditional notion of portability. Upward-compatibility establishes a partial ordering *UPCOMP* on the set of machine-classes. Programs that are executable on machines in a machine-class *M* are also executable on all machines belonging to classes that are *UPCOMP* to *M*, as long as no low-level language features are used to circumvent type-safety.

A **machine's class** is determined by the *number of bytes* it uses to represent those basic data types of the programming language that are **precise** (in contrast to the approximate *REAL*) and **infinite** (in contrast to the finite *CHAR*). In Oberon, these are the *SET* data type and the numeric types *SHORTINT*, *INTEGER*, and *LONGINT*. A machine *M'* is *UPCOMP* to a machine *M* if it represents each of these data types by at least as many bits as *M* does, and if it uses identical models of real arithmetic. Figure 6.1 illustrates this relationship for some typical processor architectures in use today. Note that neither of the two *16/32-bit* machines is *UPCOMP* to the other, but that they are both *UPCOMP* to the *16-bit* architecture.

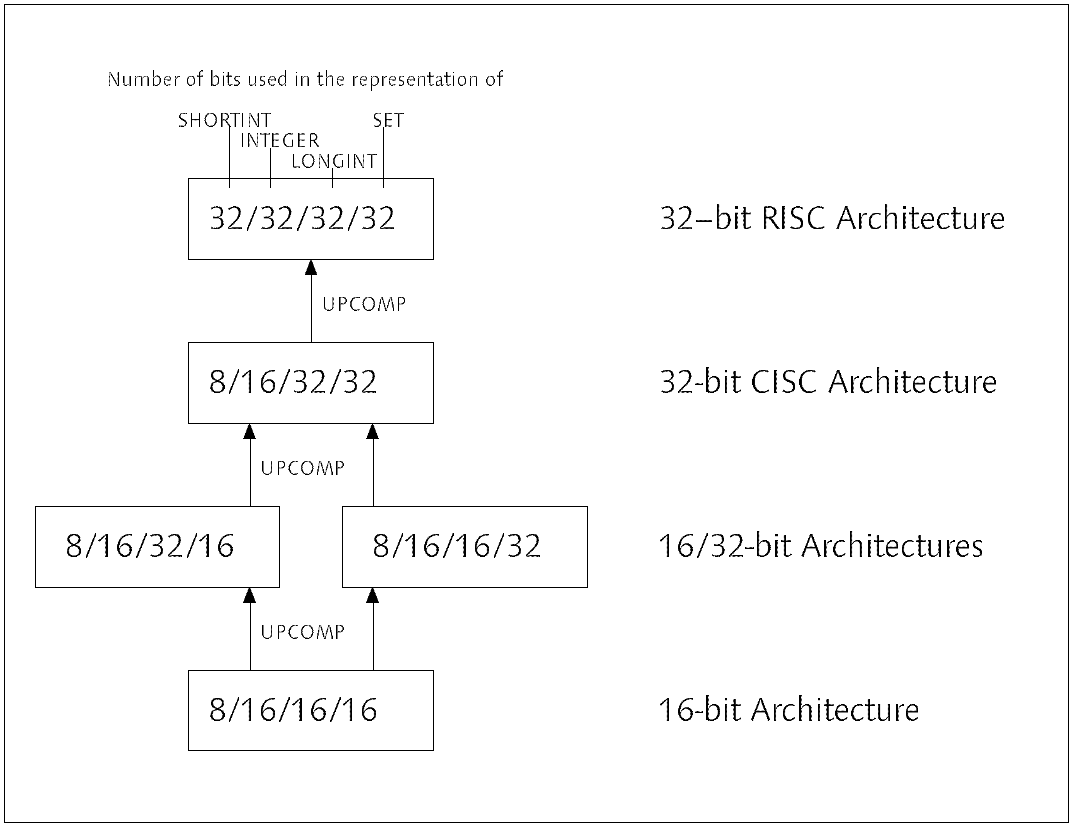


Figure 6.1: Upward-Compatibility among Different Machine-Classes

In order to decide whether a program is portable to a certain target architecture without emulation, one therefore needs information about the minimal machine-class that is required for its execution. Considering that almost all processors in use today implement the identical IEEE model of real arithmetic, this may simply be a list of prerequisite data sizes, which can be encoded in a **tag value** and appended to the program. The tag can be provided by the programmer (in which case a deep understanding of the algorithm is needed) or approximated by assuming that the program has been designed to be executable on the machine on which it originated. One then simply uses the originating machine's characteristic tag value.

Having a tag value available lets us determine in advance whether a program will be able to run at all on a certain target machine, without having to load and execute it. This should be practical in a heterogeneous environment, in which all programs are maintained in a portable intermediate representation although not all of them are executable on each of the machines.

In passing, we also observe that a cross-compiler for a machine-class *M* should only be used on a machine that is *UPCOMP* to *M*, because it may have to perform constant arithmetic in the value range of the target machine. However, the use of emulation is less critical in this case and may be tolerable during a boot-strapping process.

Software Components

At the 1968 NATO conference, during which the term of *software engineering* was coined, McIlroy [McI68] argued that "software production today appears in the scale of industrialization somewhere below the more backward construction industries" and attributed this to the absence of a **software-component industry**. Yet more than twenty-five years later, not much has changed in the way we construct software. Reuse of software at best takes place within commercial organizations, but not between them. An industrial programmer still cannot just open a catalog of standard software parts and order a module from it that will execute an algorithm according to some given specifications. At best, he can hope that the algorithm he requires is built into the operating system. Strangely enough, today we witness an ongoing standardization of software functions by incorporation into operating systems and related libraries, instead of a separate industry developing software components.

Why, then, has no independent market for platform-independent standard software components developed over the years? Why have operating systems and their supporting libraries instead grown to an awesome complexity, encompassing functions as diverse as user-interface management and data-base support? The answer may simply be that there is currently no commercial incentive to develop **plug-in software components** because the maintenance costs would be prohibitive in today's marketplace. An independent software-component vendor would have to provide his products either in a multitude of link and object formats, which is costly, or in source form, which requires complex legal arrangements to protect the intellectual property of the authors and might cost even more. On the other hand, there is a direct commercial advantage for an operating-system vendor when he adds functionality to his product. As a consequence, we see a proliferation of operating-system-level enhancements instead of an intermediate industry for operating-system-independent support libraries.

The work presented in this dissertation contributes to lowering the cost of providing drop-in software components. It demonstrates the feasibility of a platform-independent software distribution format that enables portable modules to be used right out-of-the-box, without any off-line steps of compilation or linking, *on any machine that is upward-compatible* to the machine on which the component originated.

The *on-line* aspect is important because it means that even end-users can migrate libraries in their possession to new hardware platforms. It also suggests that different component-vendors could offer competing implementations of the same library, which an end-user would install or replace simply by *plugging in*. This is analogous to the situation in the personal computer hardware market, in which end-users are expected to buy and install themselves certain parts such as floating-point co-processors.

Intra-Architectural Compatibility

Different implementations of the same architecture are beginning to diverge by so much that it is becoming increasingly difficult to generate native object-code that performs well on all processors within a family. Among other features, processors within an architecture differ in the number of instructions that can be issued simultaneously to independent functional units, and in the depth of their instruction pipelines.

Fast on-the-fly code generation presents a solution here, and may eventually even replace traditional *binary compatibility* among processors implementing the same architecture. It allows to provide each processor with a version of object code that is custom-tailored towards its particular characteristics, for example with respect to the scheduling of individual instructions, and does so even for existing programs when new processor models are introduced at a later time. One merely has to provide an updated code-generating loader attuned to the new processor variant, and all existing portable software will at once execute on the new hardware with maximum efficiency.

Since it lowers the importance of binary backward-compatibility, fast on-the-fly code-generation may in the long run also lead to a reduced coupling of hardware and software architectures, allowing the two to evolve more independently of each other.

7. Further Applications

Besides being able to provide a basis for portable software, as outlined in the previous chapter, *fast on-the-fly code generation* by a *code-generating loader* has further interesting applications. The following presentation is by no means exhaustive, but it should be sufficient to expose the potential of the new technique.

Run-Time Integrity Checks

On-the-fly code generation eliminates many of the cases that have traditionally required *several versions* of a module to coexist side-by-side on a single target machine. During development, we often need variants of standard library modules because the software being developed might not be robust enough to guarantee that all constraints of the library are fulfilled. Accordingly, a **development version** includes additional checks that validate the arguments passed to library routines. For reasons of efficiency, however, one would not want to perform these validations during regular operation when all clients of the library have been thoroughly tested and can be trusted. Consequently, the tests are usually removed from the **production version**.

For example, consider the following procedure

ReadBytes(f: File; VAR b: ARRAY OF CHAR; n: LONGINT)

in module *Files*. It is essential that the procedure never reads beyond the end of the input buffer, i.e. the precondition *n <= LEN(b)* must hold. In some cases, the compiler alone may be able to verify that this condition is satisfied, given that there is a suitable mechanism for specifying such context requirements. However, there will always be cases in which the precondition can only be verified at run-time. Therefore, a corresponding validation needs to be present in the development version of module *Files*, but not necessarily in the production version.

Unfortunately, a serious management problem arises from having to maintain more than one compiled version of the same module, and having to keep track of which one is which. One constantly has to make sure that changes in a source text are propagated to all compiled variants of the module that may exist concurrently, and use an elaborate naming scheme to differentiate between the different object files.

On-the-fly code generation does away with module variants and the associated management overhead. Only at the time of loading do we need to indicate whether we require a *development* or a *production* version of a module. The implemented system supports several independent run-time integrity guards that can be enabled selectively (Figure 7.1). Instead of the individual object file, it is then the run-time-environment that dictates whether or not these guards will be generated. Likewise, the current implementation provides for the optional insertion of routine names in the code, in a format acceptable to the standard debugger of the host machine.

Switch	Effect if Enabled
debug	annotate code for symbolic debugging
clear	initialize local pointer variables to NIL at procedure entry
nil	test pointers for NIL prior to dereferencing
index	check that array subscripts lie within bounds
type	perform run-time type tests (used with type extension)
assert	generate code for ASSERT function

Figure 7.1: Code Generation Switches

The key to **argument validation** in the implemented system lies in the standard function *ASSERT* of the programming language Oberon [Wir88]. *ASSERT* accepts as its arguments a Boolean expression and an Integer constant. It has the effect of a run-time test to check if the Boolean expression yields *TRUE*. If it doesn't, a run-time trap to the exception vector indicated by the second argument is taken. The main point about *ASSERT* is that it can be turned off by a compiler option, so that no code will be generated at all. Allowing the user to decide at run-time whether or not assertions should be verified eliminates the need for a separate development environment.

Management of Changes

Having available a system in which native code is generated only at the time of loading reduces also the organizational overhead required to keep a modular application consistent. Each time that a module is loaded, the implemented system performs a recompilation of the module's implementation, but in a manner that is completely transparent to the user. Consequently, the effects of certain changes of library modules can remain invisible, in contrast to other systems in which source-level recompilations of client modules are unavoidable.

A module needs to be recompiled whenever its own implementation is changed, or whenever it is invalidated by a change in one of the modules it depends on. Deciding which clients are invalidated by a change in a library is the difficult part. The easiest solution, implemented in tools such as the *Make* utility [Fel79] of the UNIX operating system [TR74], is to invalidate *all clients* each time that a library is changed. However, this introduces many recompilations that could be avoided in principle.

Tichy and Baker [TB85, Tic86] have introduced the notion of **smart recompilation**. This technique is founded on a detailed analysis of import and export relationships, considering not only each of the involved modules as a whole, but also the individual features that are exported from one module and imported by another. The results of this analysis are maintained in a database, along with a module dependency graph. One can then mechanize the decision which modules need to be recompiled by comparing the *changed feature set* of a modified library with the *referenced feature sets* of clients. In some cases, such as the addition of further procedures to a library, an interface change need not invalidate all existing clients then.

The proposed method opens a path to even further reductions in the number of (source-text) recompilations. A large proportion of changes that typically occur during the software life-cycle has no effect on the *behavior* of a program but nevertheless on the machine code being generated, because native code contains addresses, sizes, and relative offsets *literally*, and addressing modes often depend on particular address values. An example for a change that has consequences only in the code generator is the *insertion of an additional data field* in an exported record type. Apart from the possible error that may occur if the corresponding identifier is used already within the scope of the record, a condition that is detected easily, this addition preserves the semantics of all client modules. Unfortunately, however, the addition alters the size of the record type, and may change the relative offsets of some of the existing record fields, so that new code needs to be generated for all modules that use the record type. In a system such as the implemented one, code generation happens transparently and need not be of concern to users, because all client modules will be updated automatically when they are loaded the next time.

In principle, therefore, in a system in which code generation occurs at loading time, recompilation (of source code) is *not required to propagate changes*. There are, of course, certain changes that invalidate the source code of some client modules completely, such as removing a routine from a library module or changing the result type of a library function, but these require some *source-level re-coding* of all affected clients and cannot simply be dealt with by simple recompilation. Such situations can be detected in advance using the techniques described by Tichy and Baker, or will in any event be flagged when the code-generating loader senses an interface mismatch, resulting in a load error.

Accordingly, in a system offering on-the-fly code generation at load time, the only factor that determines whether (source-text) recompilation of clients is necessary in reaction to changes in libraries, is the strategy that is used for describing the inter-module links in the symbol table. The current implementation uses *per-module fingerprints* to ensure interface consistency, so that more recompilations are needed than would be necessary if *per-feature fingerprints* were used. In the latter case, no recompilations of clients would be necessary ever. However, this aspect has not been the main focus of the current work and, therefore, not been pursued.

Improving Code Quality by Targeted Optimizations

The fact that object code is generated anew each time that a module is loaded could also be exploited for increasing the *overall performance* of the whole system, although this has currently not been implemented. Borrowing from ideas discussed by Morris [Mor91] and Wall [Wal91], an **execution profile** obtained in a previous run of the system could guide the level of optimization applied by the code-generating loader in the creation of the next executable version.

While fine-grained profiling might be useful as a basis for specifically-targeted optimizations, run-time profiling, which is associated with an overhead, might not even be necessary in a modular system. Instead, one might simply use the *import counter*, which indicates how many clients a module has, as an estimate for the "relative importance" of a module. The more important a module is, the greater the potential benefits of optimization.

The idea of targeted optimization may be developed even further, taking into account that object code can be re-created from SDE-files at any time. The system might expend its *idle time* on the recompilation of whole subtrees of the loaded-module graph, employing a higher optimization level than the currently loaded version. After recompiling such a subtree, it may then attempt to unload the old version, and if unloading is successful adjust the global module graph to include the new, optimized version. Note that the unloading step may fail if further clients are added to the originally loaded modules while re-generation is underway, or if installed procedures from the original modules remain active in the system.

Further Applications

A machine-independent abstract program representation from which high-quality code can efficiently be generated on-the-fly might also prove to be valuable in the context of heterogeneous distributed systems consisting of several different hardware platforms.

For example, the proposed technique could form the basis for a very general **remote procedure-call** mechanism [Nel81]. Instead of compiling a separate stub for each procedure to be called remotely and installing it as a process on the target machine, one might send a complete instruction sequence in a machine-independent format, to be processed by a single *code-generating stub* on the side of the receiver. Cryptological authentication measures could be applied to prevent misuse in an open network.

Consistency problems between program segments for different machines in a distributed application could be avoided trivially by sending a **consistent version** across the network before the start of a distributed computation, thereby guaranteeing that identical code executes on all machines taking part in the computation.

Last but not least, designers of object-oriented systems could use an even broader definition of **object persistence**. A persistent object might contain its own code in an abstract format. Such an object could then migrate over a network or be transported on some storage medium. At a destination site, first the code to handle the object would be generated dynamically. Thereafter, the object's data would be read in.

8. Related Work

The project described in this thesis was started in 1990, and first results were published in September of 1991 [FL91]. At that time, it seemed almost exotic to attempt any revival of the old *UNCOL* idea. Today, the topic again seems "hot", and many researchers are working on related projects. The most notable of these other projects is the *architecture neutral distribution format (ANDF)* initiative by the *Open Software Foundation (OSF)*, which will be discussed in the first part of this chapter. The remaining paragraphs mention other topics that are related to the subject of this thesis in certain ways.

The OSF ANDF Project

The Open Software Foundation (OSF) is a not-for-profit organization jointly established by several companies in the information technology industry. In May 1989, OSF solicited proposals for an **architecture neutral software distribution format** (ANDF). They received twenty-three such proposals in response, and in June 1991, OSF selected a technology called *TDF* [OSF91, DRA93a, DRA93b], designed and implemented by the United Kingdom Defence Research Agency (DRA), to serve as the basis of their ANDF. In the meantime, TDF has been adopted also by UNIX System Laboratories and the European Community's Esprit Program.

The technology of TDF is still developing. With one exception [Bra92], all existing documentation about TDF at the time of this writing still comes from DRA and the OSF. Unfortunately, this documentation does not describe the current state of the TDF development very accurately, as can be inferred from a footnote in [DRA93d]:

> *"The description of the TDF system in this paper reflects a very slightly idealised version of the current technology; one in which all the features we intend to put in, but have not had time to, are included."*

TDF has many characteristics in common with semantic-dictionary encoding. Just like semantic-dictionary encoding, and unlike previous *UNCOL* attempts, TDF is not based on an abstract machine, but on a *tree-structured* intermediate language in conjunction with an embedded symbol table.

Scope of TDF versus that of Semantic-Dictionary-Encoding

TDF has been designed to be both source-language and target-architecture independent, although as of June 1993, the only existing compiler front-end for TDF was for the programming language C [KR78]. TDF is claimed to be useful for source languages other than C, and compilers translating into the TDF representation from other languages are being developed. In regard to the suitability of TDF for encoding programs in other languages, the official TDF documentation [DRA93c] states the following:

> *"TDF constructs have been carefully designed so as to be able to accommodate the particular variants found in different programming languages. However, TDF cannot guarantee coverage of new programming languages as it can for new architectures. New languages might contain novel features that are not efficiently implementable using existing features of TDF. [...] It is likely that the implementation of [TDF encoders] for programming languages other than C will expose new features that would enhance the efficiency of the run-time code for these new languages – it is likely that such extensions can be added in an upwards compatible manner, but this cannot be guaranteed."*

In contrast, the method of SDE is not a program representation in its own right, but a **meta-technique** for encoding programs abstractly. It is parametrized by the initial configuration of the dictionary and the heuristics used for dictionary management.

So far, SDE has been applied only to programs originally written in the programming language Oberon [Wir88]. However, since semantic meaning is instilled into each SDE-file solely by the *initial configuration* of a dictionary, it is possible to support easily future language requirements, even without invalidating existing SDE-files in an old format. All that is necessary is a key in the SDE-file that uniquely identifies the initial configuration of the dictionary that has to be used for decoding. This might, for example, be simply the name of a file in which the configuration is stored.

Hence, SDE allows us to **evolve the set of encodeable language constructs** independently of the actual file-formats. In fact, by using configuration files, individual SDE-decoders could be made completely independent of the file formats they need to process. It would require a standardization only of the *meanings of different meta-language constructs*. Software developers would then be able to choose freely which of these meta-language constructs to use in the encoding of their programs, and at which positions of the initial dictionary they would place these constructs. The smaller the set of meta-language constructs, the more difficult it will be to reverse-engineer the encoded program, although the use of fewer constructs could also affect compactness and optimizeability adversely.

Ease of Reverse-Engineering

The success of any new technology will lastly depend on its acceptance in the marketplace. In the case of a machine-independent software-distribution format, software vendors need to be convinced that they do not give away their trade secrets when distributing their products in this way. This conviction will not be brought about easily.

In effect, any intermediate format that preserves the abstract structure of programs can be reverse-engineered to produce a "shrouded" source program, i.e. one that contains no meaningful internal identifiers [Mac93]. However, with current technology, reverse-engineering to a similar degree is possible also from binary code. Many of the algorithms that have been developed for object-code-level optimization [DF84] are useful for these purposes.

Moreover, the statement [DRA93c] is probably correct that portable formats are such attractive targets to reverse-engineer that suitable tools will become available anyway, regardless of how difficult it is to produce such tools. It would, therefore, not make much sense to jeopardize the advantages of SDE in an attempt to make reverse-engineering more difficult.

Performance of SDE in Comparison to TDF

SDE has some performance-advantages over TDF. To start with, it is more compact. Although in [OSF91] the OSF recognized that *"it is important that the ANDF file size be as small as possible"*, TDF is in fact less compact than object code. TDF is quoted [DRA93c] as being "around twice the size of the binary of CISC machines", while SDE is at most half the size of MC68020 binary code.

Because of the lack of a common operating platform on which both mechanisms have been implemented, a direct performance comparison of SDE versus TDF is currently not possible. However, DRA [DRA93c] state that native object-file generation takes between 32% and 83% of native C compile time. Considering that Oberon compilers usually outperform C compilers by a factor of more than 15 in compilation speed [BCF92], and that loading of SDE-files is more than four times faster than regular Oberon compilation, it seems reasonable to claim that, at least on CISC processors and without taking code quality into account, SDE should provide for much faster code generation than TDF.

In TDF, the individual modules (called *capsules* in TDF terminology) that comprise an application program are linked together statically and then translated into native code in an *off-line* process. Conversely, SDE has been designed to support *dynamic module loading* with on-the-fly code-generation. As has been explained in Chapter 6, only the latter approach leads to user-serviceable, plug-in software *components*.

Incremental Compilation and Linking

In some ways, the implemented could also be compared with systems offering incremental compilation or incremental linking. After all, whenever a new module is installed by the code-generating loader, this amounts to a compilation and a linking step, in the course of which the code-base of the executing environment is increased.

Incremental compilers are highly complex programs that are often integrated with structural editors [EC72, MF81, TR81, FS84], and optimizing variants [PS84, PS92] are of even more formidable intricacy. In light of the speed of current code generators it is highly questionable whether the technology of incremental compilation should be developed any further. **Separate compilation** of modular programs seems to be a far simpler means of achieving the same goal.

On the other hand, **incremental linking** systems, such as the one by Quong and Linton [QL91], require large amounts of memory and generate large object files, in which extra space is allocated between modules in order to allow the code to grow between versions. If a module grows beyond the space reserved for it, then the module following it on the object file has to be re-linked as well, and this *overflow-effect* may propagate all the way to the end of the object file. Without doubt, the technique of **dynamic loading**, upon which my own method is based, is more elegant and efficient than incremental linking.

I believe that both incremental compilation and incremental linking will cease to be of any significance as code generators become ever faster. Instead, separate compilation of program modules should gain in importance, aided by the spread of modular programming languages and the availability of module loaders that can process machine-independent object formats directly. This will give vital new impulses to the field of library design and increase productivity by way of software reuse.

Dynamic Translation

The concept of **dynamic translation** of programs from one representation into another has been around for some time. Early implementations, such as one by Brown [Bro76], were developed with the aim of balancing execution speed and memory requirements under the extreme hardware constraints that were then the norm. These early implementations applied only to programming languages without block-structure and performed the generation of native object-code on a statement-by-statement basis.

For a long time, the main reason for implementing dynamic translation remained the fact that it allowed an elegant trade-off between execution efficiency and memory consumption. A paper by Rau [Rau78] classifies program representations into three categories, namely *high-level*, *directly interpretable*, and *directly executable*, and discusses the use of dynamic translation between these categories as a means for achieving speed and compactness simultaneously.

Then came Deutsch and Schiffmann's [DS84] landmark paper on the efficient implementation of the programming language Smalltalk-80 [GR83]. They used dynamic translation for increasing execution speed while *retaining virtual-machine code-compatibility* with existing implementations. The latter was necessary because the Smalltalk-80 virtual machine is actually visible to user programs, and much of the system code depends on it. In Deutsch and Schiffmann's implementation, native code for the actual target machine is generated on-the-fly and cached until it is invalidated by changes in the source program, or until it is overwritten in the code-cache due to lack of space.

A more recent application of dynamic translation comes from the implementation [CUL89, CU89, CU90, CU91] of the programming language *Self* [US87], a dynamically-typed language based on prototypes. Just as the Smalltalk-80 system, it can benefit enormously from on-the-fly code generation, because type information, although unavailable statically, is available at run-time. By allowing the dynamic generation of several variants of an expression, optimized for different run-time types of the component variables, the efficiency of such systems can be multiplied, but still cannot compete with statically-typed programming languages.

In contrast, the implemented system attempts to deliver run-time performance comparable to traditional compilers and offers on-the-fly code generation primarily as a means for increased user convenience, not code quality. It ties dynamic translation intimately to the two concepts of *separate compilation* and *dynamic module loading*. The information contained in SDE-files is equivalent to the source description, so that there is no fundamental limit to the obtainable code quality. Hence, the most effective optimizing code generators could potentially be built into a code-generating loader operating on SDE.

It is also true that *modules* are much better suited as the unit of code generation than procedures. A module is a collection of data types, variables, and procedures that are loaded together always, and, almost equally important, unloaded together always. Furthermore, a module can be loaded only after all of its servers (imported library modules) have been loaded successfully, and unloaded only after all of its clients (importing modules) have been unloaded. Consequently, code is generated from the bottom upwards, and the addresses of all callees are known when compiling a caller. Likewise, there are never any clients that need to be invalidated explicitly when a module is unloaded.

In systems in which the unit of code generation is the *procedure*, such as Smalltalk-80 and Self, program execution is interspersed with code-generation. Each procedure call may potentially fault, at which point native code needs to be generated dynamically before the call can be completed. Unfortunately, this may sometimes generate formidable amounts of such faults in succession, each associated with a re-load of the instruction cache, causing disruptive delays for interactive users at unexpected points in time.

Dynamic Binary-To-Binary Object-Code Translation

A technology that has emerged only recently is **binary translation of object code** [SCK93]. It enables programs for one architecture to be executed directly on another, by way of true translation of whole object programs into the native instruction set of the new target machine. The main rationale behind binary translation is the need to protect previous investments into software when migrating to a new hardware architecture. Rather than constituting a *portability technique* in the spirit of "software components", binary translation represents a *capitulation* before the fact that much of the software in existence is not portable, and cannot be ported by ordinary means; for example, because the source texts and the original design documents are no longer available.

Binary translation is a complex technique. Since it is put to use mainly in circumstances in which little is known about the programs that serve as its input, the translation mechanism needs to be suitable for any program that could possibly have executed on the architecture of origin. This includes programs that modify themselves. Consequently, self-modification conditions need to be detected on the new target machine, at which time the affected code segments may have to be re-translated on-the-fly.

Due to its complexity, binary translation cannot really be seen as an answer to the portability problem, but only as an intermediate solution allowing us to keep using an existing software base while it is being rewritten for the new architecture. The technique that is being advocated in this dissertation is much simpler and concerned less with backward-compatibility than with forward-compatibility, with whatever may lie ahead in the future.

Syntax-Directed Source Compression

SDE utilizes the syntactic structure of programs to achieve a high information density. Hence, the subject of this dissertation is distantly related to previous work in the area of **syntax-directed source compression**. Only two implementations are documented in the literature:

- Contla [Con85] describes a program-source-compaction technique that is based on *recording the path followed by the parser* while traversing the syntax tables of the programming language in which the program is written. For this purpose, the alternatives in each production of the language grammar are labelled. A syntactically correct program can then be represented by the sequence of particular choices that need to be taken in the process of deriving the program from the start symbol of the grammar.

  Programs in such a grammar-path representation are decoded by a parser without any look-ahead, in which the next action to be taken follows directly from the front-most encoded symbol on the input stream. Although such a decoder is simpler than an ordinary source-text parser, it is a parser nevertheless. Generating object code directly from Contla's representation would therefore not be much different from ordinary compilation, apart from possible speed gains due to reduced input overhead. Contla himself not even hints at this possibility, perceiving his work strictly as a storage reduction method.

- Katjainen et al. [KPT86] report of a program-source-compression method based on parsing, which has been implemented for the programming language Pascal [Wir71]. They represent programs by a pre-order enumeration of the parse tree, along with an encoding of the symbol table. However, for purported reasons of efficiency (their system is written in Prolog [CM84]), they do not generate semantically correct parse trees for expressions, but parse them as if there were no operator precedence. This is of course permissible if textual reproducibility of source programs is the only objective. Before code can be generated from their representation, however, it has to be fully expanded and re-parsed. The possibility of immediate object-code generation directly from the compact representation was, obviously, not anticipated by the designers of this data compression method.

Both of these techniques are concerned only with preserving the source text of a program in a compact form. SDE goes beyond that objective, additionally striving to represent the program's semantic content in a way that is well suited for dynamic code-generation, and achieves this goal admirably.

9. Summary and Conclusion

This dissertation has described a new technique called *semantic-dictionary encoding* (SDE) for representing programs abstractly. SDE yields a highly compact encoding and is able to provide a code generator with all the information available on the level of the source language, plus additional knowledge about the occurrence of common subexpressions in the source text. It facilitates simple and efficient on-the-fly code generation on relatively slow processors without precluding the use of highly optimizing code generation methods on faster ones.

The new technique is able to provide separate compilation of program modules with type-safe interfaces, independent module distribution, and interchangeability of modules with the same interface. These properties give it an advantage over other approaches to portability, such as using a "shrouded" high-level programming language. While it is true that many of the cost-lowering benefits of portability can be gained by using shrouded source-code, code generation from the SDE representation is also much faster than ordinary compilation

The proposed technique offers the potential of providing a universal software distribution format that is practical to use. Hence, it might encourage software developers to share reusable software components or offer them commercially. It eliminates the need for separate development environments and can reduce the number of recompilations after changes in library modules. Other potential applications of object-level portability may lie in heterogeneous distributed computing environments, in which the compactness of the SDE representation would be of particular advantage when network transfer is required.

References

[Air76] T. J. Aird; The IMSL Fortran Converter: An Approach to Solving Portability Problems; *Workshop on the Portability of Numerical Software*, published as *Springer Lecture Notes in Computer Science*, 57, 368–388; 1976. {6}

[App85] Apple Computer, Inc.; *Inside Macintosh*; Addison-Wesley; 1985ff. {4}

[Atk77] M. P. Atkinson; IDL: A Machine-independent Data Language; *Software–Practice and Experience*, 7:6, 671–684; 1977. {6}

[ADH89] R. Atkinson, A. Demers, C. Hauser, Ch. Jacobi, P. Kessler and M. Weiser; Experiences Creating a Portable Cedar; *Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation*, published as *Sigplan Notices*, 24:7, 322–329; 1989. {2}

[BKM87] A. B. Bergh, K. Keilman, D. J. Magenheimer and J. A. Miller; HP 3000 Emulation on HP Precision Architecture Computers; *Hewlett-Packard Journal*, 38:11, 87–89; 1987. {6}

[Boy76] J. M. Boyle; Mathematical Software Transportability Systems: Have the Variations a Theme?; *Workshop on the Portability of Numerical Software*, published as *Springer Lecture Notes in Computer Science*, 57, 304–353; 1976. {6}

[BD74] J. M. Boyle and K. W. Dritz; An Automated Programming System to Facilitate the Development of Quality Mathematical Software; *Information Processing 74* (Proceedings of the IFIP Congress 74), North Holland, 542–546; 1974. {6}

[BCF92] M. Brandis, R. Crelier, M. Franz and J. Templ; *The Oberon System Family*; Report #174, Departement Informatik, ETH Zurich; 1992. {3, 8}

[Bra92] M. Brandreth; All for One and One for All; *Physics World*, 5:6, 47–50; 1992. {8}

[Bro72] P. J. Brown; Levels of Language for Portable Software; *Communications of the ACM*, 15:12, 1059–1062; 1972. {6}

[Bro76] P. J. Brown; Throw-Away Compiling; *Software–Practice and Experience*, 6:3, 423–434; 1972. {8}

[BH76] W. S. Brown and A. D. Hall; Fortran Portability via Models and Tools; *Workshop on the Portability of Numerical Software*, published as *Springer Lecture Notes in Computer Science*, 57, 158–164; 1976. {6}

[CU89] C. Chambers and D. Ungar; Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language; *Proceedings of the ACM Sigplan '89 Conference on Programming Language Design and Implementation*, published as *Sigplan Notices*, 24:7, 146–160; 1989. {8}

[CU90] C. Chambers and D. Ungar; Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs; *Proceedings of the ACM Sigplan '90 Conference Programming Language Design and Implementation*, published as *Sigplan Notices*, 25:6, 150–162; 1989. {8}

[CU91] C. Chambers and D. Ungar; Making Pure Object-Oriented Languages Practical; *OOPSLA '91 Conference Proceedings*, published as *Sigplan Notices*, 26:11, 1–15; 1989. {8}

[CUL89] C. Chambers, D. Ungar and E. Lee; An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes; *OOPSLA '89 Conference Proceedings*, published as *Sigplan Notices*, 24:10, 49–70; 1989. {8}

[Chu35] A. Church; An Unsolvable Problem of Elementary Number Theory; *American Journal of Mathematics*, 58, 345–363; 1936. {2}

[CM84] W. F. Clocksin and C. S. Mellish; *Programming in Prolog: Second Edition*; Springer; 1984. {8}

[CPW74] S. S. Coleman, P. C. Poole and W. M. Waite; The Mobile Programming System, Janus; *Software–Practice and Experience*, 4:1, 5–23; 1974. {2}

[Con58] M. E. Conway; Proposal for an UNCOL; *Communications of the ACM*, 1:10 5–8; 1958. {2}

[Con85] J. F. Contla; Compact Coding of Syntactically Correct Source Programs; *Software–Practice and Experience*, 15:7, 625–636; 1985. {8}

[Cre91] R. Crelier; OP2: A Portable Oberon-2 Compiler; *Proceedings of the 2nd International Modula-2 Conference*, Loughborough, England, 58–67; 1991. {3}

[Dah84] I. Dahlstrand; *Software Portability and Standards*; Ellis Horwood, Chichester; 1984. {6}

[DF84] J. W. Davidson and C. W. Fraser; Code Selection through Object Code Optimization; *ACM Transactions on Programming Languages and Systems*, 6:4, 505–526; 1984. {8}

[DRA93a] United Kingdom Defence Research Agency; *TDF Specification, Issue 2.1*; June 1993. {2, 8}

[DRA93b] United Kingdom Defence Research Agency; *A Guide to the TDF Specification, Issue 2.1.0*; June 1993. {8}

[DRA93c] United Kingdom Defence Research Agency; *Frequently Asked Questions about ANDF, Issue 1.1*; June 1993. {8}

[DRA93d] United Kingdom Defence Research Agency; *TDF and Portability, Issue 1.0*; June 1993. {8}

[DS84] L. P. Deutsch and A. M. Schiffmann; Efficient Implementation of the Smalltalk-80 System; *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, 297–302; 1984. {8}

[Dig92] Digital Equipment Corporation; *Alpha Architecture Handbook*; 1992. {6}

[DMB79] J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart; *LINPACK Users' Guide*; Society for Industrial and Applied Mathematics, Philadelphia; 1979. {6}

[Dri76] K. W. Dritz; Multiple Program Realizations using the TAMPR System; *Workshop on the Portability of Numerical Software*, published as *Springer Lecture Notes in Computer Science*, 57, 405–423; 1976. {6}

[EC72] J. Earley and P. Caizergues; A Method for Incrementally Compiling Languages with Nested Statement Structure; *Communications of the ACM*, 15:12, 1040–1044; 1972. {8}

[Fel79] S. I. Feldman; Make: A Program for Maintaining Computer Programs; *Software–Practice and Experience*, 9:4, 255–265; 1979. {7}

[Fox76] P. A. Fox; Port: A Portable Mathematical Subroutine Library; *Workshop on the Portability of Numerical Software*, published as *Springer Lecture Notes in Computer Science*, 57, 163–177; 1976. {6}

[FS84] R. Ford and D. Sawamiphakdi; A Greedy Concurrent Approach to Incremental Code Generation; *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, 165–178; 1985. {8}

[Fra90a] M. Franz; *The Implementation of MacOberon*; Report #141, Departement Informatik, ETH Zürich; 1990. {3, 4}

[Fra90b] M. Franz; *MacOberon Reference Manual*; Report #142, Departement Informatik, ETH Zurich; 1990. {3, 4}

[Fra91] M. Franz; The Rewards of Generating True 32-bit Code; *Sigplan Notices*, 26:1, 121–123; 1991. {3}

[Fra93a] M. Franz; Emulating an Operating System on Top of Another; *Software–Practice and Experience*, 23:6, 677–692; June 1993. {3, 4, 6}

[Fra93b] M. Franz; The Case for Universal Symbol Files; *Structured Programming*, 14:3, 136–147; October 1993. {2, 3, 4}

[FL91] M. Franz and S. Ludwig; Portability Redefined; *Proceedings of the 2nd International Modula-2 Conference*, Loughborough, England; 1991. {6, 8}

[GF84] M. Ganapathi and C. N. Fischer; Attributed Linear Intermediate Representations for Retargetable Code Generators; *Software–Practice and Experience*, 14:4, 347–364; 1984. {2}

[Gol84] A. Goldberg; *Smalltalk-80: The Interactive Programming Environment*; Addison-Wesley; 1984. {4}

[GR83] A. Goldberg and D. Robson; *Smalltalk-80: The Language and its Implementation*; Addison-Wesley; 1983. {8}

[Gri72] R. E. Griswold; *The Macro Implementation of SNOBOL4: A Case Study in Machine-Independent Software Development*; Freeman, San Francisco; 1972. {2}

[Gri78] R. E. Griswold; A History of the SNOBOL Programming Languages; in R. L. Wexelblat, editor, *History of Programming Languages (Proceedings of the of the History of Programming Languages Conference)*, Academic Press (ACM Monograph Series), New York, 601–645; 1981. {2}

[HW78] B. K. Haddon and W. M. Waite; Experience with the Universal Intermediate Language Janus; *Software–Practice and Experience*, 8:5, 601–616; 1978. {2}

[HF76] S. J. Hague and B. Ford; Portability: Prediction and Correction; *Software–Practice and Experience*, 6:1, 61–69; 1976. {6}

[Hal82] J. A. Hall; A Microprogrammed P-Code Interpreter for the Data General Eclipse S/130 Minicomputer; *Software–Practice and Experience*, 12:8, 755–765; 1982. {2}

[Hal65] M. I. Halpern; Machine Independence: Its Technology and Economics; *Communications of the ACM*, 8:12, 782–785; 1965. {2}

[IEEE85] *IEEE Standard for Binary Floating-Point Arithmetic*; ANSI/IEEE Standard 754; 1985. {6}

[KPT86] J. Katajainen, M. Penttonen and J. Teuhola; Syntax-directed Compression of Program Files; *Software–Practice and Experience*, 16:3, 269–276; 1986. {8}

[KR78] B. W. Kernighan and D. M. Ritchie; *The C Programming Language*; Prentice-Hall; 1978. {2, 8}

[KKM80] P. Kornerup, B. B. Kristensen and O. L. Madsen; Interpretation and Code Generation Based on Intermediate Languages; *Software–Practice and Experience*, 10:8, 635–658; 1980. {2}

[Kro76] F. T. Krogh; *A Method for Simplifying the Maintenance of Software which Consists of Many Versions*; Memorandum 314, Jet Propulsion Laboratory, Pasadena, California; 1976. {6}

[Mac93] S. Macrakis; *Protecting Source Code with ANDF*; Open Software Foundation Research Institute; June 1993. {8}

[McI68] M. D. McIlroy; Mass Produced Software Components; in Naur, Randell, Buxton (eds.), *Software Engineering: Concepts and Techniques*, Proceedings of the NATO Conferences, New York, 88–98; 1976. {6}

[MF81] R. Medina-Moria and P. H. Feiler; An Incremental Programming Environment; *IEEE Transactions on Software Engineering*, 7:5, 472–482; 1981. {8}

[Mor91] W. G. Morris; CCG: A Prototype Coagulating Code Generator; *Proceedings of the ACM Sigplan '91 Conference on Programming Language Design and Implementation*, published as *Sigplan Notices*, 26:6, 45–58; 1991. {7}

[Mot87] Motorola, Inc.; *M68030 Enhanced 32-bit Microprocessor User's Manual*; Motorola Customer Order No. MC68020UM/AD; 1987. {3, 4, 5}

[NS84] National Semiconductor Corporation; *Series 32000: Instruction Set Reference Manual*; National Semiconductor Customer Order No. NSP-INST-REF-M, Publication Number 420010099-001B, Santa Clara, California; 1984. {2}

[Nel81] B. J. Nelson; *Remote Procedure Call*; Report #CLS-81-9, Palo Alto Research Center, Xerox Corporation, Palo Alto, California; 1981. {7}

[NPW72] M. C. Newey, P. C. Poole and W. M. Waite; Abstract Machine Modelling to Produce Portable Software: A Review and Evaluation; *Software–Practice and Experience*, 2:2, 107–136; 1972. {2}

[NAJ76] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and Ch. Jacobi; Pascal-P Implementation Notes; in D.W. Barron, editor; *Pascal: The Language and its Implementation*; Wiley, Chichester; 1981. {2}

[OSF91] Open Software Foundation; *OSF Architecture-Neutral Distribution Format Rationale*; 1991. {8}

[PS84] L. L. Pollock and M. L. Soffa; Incremental Compilation of Locally Optimized Code; *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, 152–164; 1985. {8}

[PS92] L. L. Pollock and M. L. Soffa; Incremental Global Reoptimization of Programs; *ACM Transactions on Programming Languages and Systems*, 14:2, 173–200; 1992. {8}

[PW69] P. C. Poole and W. M. Waite; Machine Independent Software; *Proceedings of the ACM 2nd Symposium on Operating System Principles*, Princeton, New Jersey; 1969. {2}

[QL91] R. W. Quong and M. A. Linton; Linking Programs Incrementally; *ACM Transactions on Programming Languages and Systems*, 13:1, 1–20; 1991. {8}

[Rau78] B. R. Rau; Levels of Representation of Programs and the Architecture of Universal Host Machines; *Proceedings of the 11th Annual Microprogramming Workshop*, Pacific Grove, California, 67–79; 1978. {8}

[Rei89] M. Reiser; Private Communication; 1989. {1}

[Ric71] M. Richards; The Portability of the BCPL Compiler; *Software–Practice and Experience*, 1:2, 135–146; 1971. {2}

[Ric69] M. Richards; BCPL: A Tool for Compiler Writing and System Programming; *Proceedings of the 1969 Spring Joint Computer Conference*, published as *AFIPS Conference Proceedings*, 34, AFIPS Press, Montvale; 1969. {2}

[RW79] M. Richards and C. Whitby-Strevens; *BCPL: The Language and its Compiler*; Cambridge University Press; 1979. {2}

[RH77] M. G. Richardson and S. J. Hague; The Design and Implementation of the NAG Master Library File System; *Software–Practice and Experience*, 7:1, 127–137; 1977. {6}

[Ryd74] B.G. Ryder; The PFORT Verifier; *Software–Practice and Experience*, 4:4, 359–377; 1974. {6}

[ST73] E. H. Sibley, and R. W. Taylor; A Data Definition and Mapping Language; *Communications of the ACM*, 16:12, 750–759; 1973. {6}

[SCK93] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks and S. G. Robinson; Binary Translation; *Communications of the ACM*, 36:2, 69–81; February 1993. {8}

[Ste60] T. B. Steel; UNCOL: Universal Computer Oriented Language Revisited; *Datamation*, 6:1, 18–20; 1960. {2}

[Ste61a] T. B. Steel, Jr.; A First Version of UNCOL; *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, Los Angeles, California, 371–378; 1961. {2}

[Ste61b] T. B. Steel, Jr.; UNCOL: The Myth and the Fact; *Annual Review in Automatic Programming*, 2, 325–344; 1961. {2}

[SWT58] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock and T. B. Steel; The Problem of Programming Communication with Changing Machines: A Proposed Solution: Report of the Share Ad-Hoc Committee on Universal Languages; *Communications of the ACM*, 1:8, 12–18, and 1:9, 9–15; 1958. {2}

[Szy92] C. A. Szyperski; Write-ing Applications: Designing an Extensible Text Editor as an Application Framework; *Proceedings of the 7th International Conference on the Technology of Object-Oriented Languages and Systems (TOOLS'92)*, Dortmund, Germany, 247–261; 1992. {5}

[TKB78] A. S. Tanenbaum, P. Klint, and W. Bohm; Guidelines for Software Portability; *Software–Practice and Experience*, 8:6, 681–698; 1978. {6}

[TR81] T. Teitelbaum and T. Reps; The Cornell Program Synthesizer: A Syntax-Directed Programming Environment; *Communications of the ACM*, 24:9, 563–573; 1981. {8}

[TR74] K. Thompson and D. M. Ritchie; The UNIX Time-Sharing System; *Communications of the ACM*, 17:2, 1931–1946; 1974. {7}

[Tic86] W. F. Tichy; Smart Recompilation; *ACM Transactions on Programming Languages and Systems*, 8:3, 273–291; 1986. {7}

[TB85] W. F. Tichy and M. C. Baker; Smart Recompilation; *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, 236–244; 1985. {7}

[Tur36] A. M. Turing; On Computable Numbers, with an Application to the Entscheidungsproblem; *Proceedings of the London Mathematical Society*, 2nd Series, 42, 230–265; 1937. {2}

[US87] D. Ungar and R. B. Smith; Self: The Power of Simplicity; *OOPSLA '87 Conference Proceedings*, published as *Sigplan Notices*, 22:12, 227–242; 1987. {8}

[Wai75] W. M. Waite; Hints on Distributing Portable Software; *Software–Practice and Experience*, 5, 295–308: 1975. {6}

[Wal91] D. W. Wall; Predicting Program Behavior Using Real or Estimated Profiles; *Proceedings of the ACM Sigplan '91 Conference on Programming Language Design and Implementation*, published as *Sigplan Notices*, 26:6, 59–70; 1991. {7}

[Wal82] P. J. L. Wallis; *Portable Programming*; Macmillan, London; 1982. {6}

[WDH89] M. Weiser, A. Demers and C. Hauser; The Portable Common Runtime Approach to Interoperability; *Proceedings of the 12th ACM Symposium on Operating System Principles*, published as *Operating System Reviews*, 23:5, 114–122; 1989. {6}

[Wel84] T. A. Welch; A Technique for High-Performance Data Compression; *IEEE Computer*, 17:6, 8–19; 1984. {2}

[WD79] Western Digital Corporation; *Pascal Microengine*; 1979. {2}

[WMP69] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck and C. H. A. Koster; Report on the Algorithmic Language Algol 68; *Numerische Mathematik*, 14, 79–218; 1969. {2}

[Wir71] N. Wirth; The Programming Language Pascal; *Acta Informatica*, 1:1, 35–63; 1971. {2}

[Wir88] N. Wirth; The Programming Language Oberon; *Software–Practice and Experience*, 18:7, 671–690; 1988. {2, 3, 4, 7, 8}

[WG89] N. Wirth and J. Gutknecht; The Oberon System; *Software–Practice and Experience*, 19:9, 857–893; 1989. {4}

[WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992. {4, 5}