HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports — ISSN 0783–5396
No. **42**; January 1997 — ISBN 951–22–3439–4

TOWARDS OPTIMIZING CODE GENERATION BY DOMAIN-SENSITIVE MACRO EXPANSION

Eero Lassila

Digital Systems Laboratory
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, FINLAND

Helsinki University of Technology
Department of Computer Science and Engineering
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **42**; January 1997

ISSN 0783–5396
ISBN 951–22–3439–4

Towards Optimizing Code Generation by Domain-Sensitive Macro Expansion

Eero Lassila

Abstract: Many modern code generation methods use tree pattern matching with dynamic programming. However, especially in the case of an irregular special-purpose processor architecture their lack of transparency and stability may be problematic: it is difficult to predict the exact code generation result in advance, and the effects of a modification in the code generation rules may be surprisingly wide.

In contrast, macro expansion techniques are intuitively transparent. When global variables are disallowed, macro expansion typically has the Church-Rosser property: the final expansion result is independent of the expansion order of the individual intermediate macro calls. Besides enabling parallel implementation, order-independence means stability: the effects of modifying a macro definition are guaranteed to remain local.

The locality is actually the problem with macro expansion; code optimization is improved when an assembly language macro is sensitive to its context. For instance, it should know which registers are free and which ones contain useful values. A traditional mechanism for propagating contextual information is the use of global variables. But relying on global variables means that the expansion should take place strictly from left to right, so that order-independence is lost while it is still not possible to pass information in the opposite direction.

In this work a novel assembly-language-level macro expansion technique with both order-independence and rather general context-sensitivity is proposed. The technique supports modular and hierarchical code libraries. Both a formal model and a code generator prototype demonstrating the technique are presented.

Keywords: Code generation, code optimization, macro expansion, embedded system programming.

Printing: Libella Painopalvelu Oy; Espoo 1997

Helsinki University of Technology
Department of Computer Science and Engineering
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{09}{+358\text{–}9}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–9–465 077
E-mail: lab@saturn.hut.fi

Foreword

This work has been carried out at Digital Systems Laboratory of Helsinki University of Technology. I wish to thank Professor Leo Ojala for his continuous support and guidance; in particular, I appreciate with gratitude the opportunity to concentrate on the present subject.

To all my co-workers at the laboratory, I certainly owe a lot for the inspiring atmosphere. I am indebted especially to Dr. Johan Lilius and Dr. Ilkka Niemelä for their valuable suggestions on the coordination of my efforts.

The financial support from Helsinki Graduate School in Computer Science and Engineering, Academy of Finland, and Foundation of Technology is gratefully acknowledged.

Eero Lassila
Espoo, Finland
January 28, 1997

Contents

1 Introduction

Compiler writers and many other system programmers need to be familiar with assembly language. Assembly language programs are, of course, low-level and machine-specific, as opposed to the ones written in a language like Fortran, Pascal, or C. Furthermore, assembly languages usually lack structuring mechanisms: the code tends to be "spaghetti-like".

Many algorithms are conveniently described as data flow graphs; this holds especially in the case of digital signal processing algorithms [78], which are typical embedded computer system specifications. An essential advantage of data flow graphs is their natural composability [24]: a given subgraph, such as shown in Fig. 1, can be defined to be the implementation of a new compound block, such as shown in Fig. 2, which is then ready to be used exactly in the same way as the old predefined blocks. This composition straightforwardly carries over to hardware realizations: both discrete components and VLIW cells [52] are easily grouped into larger composite entities.

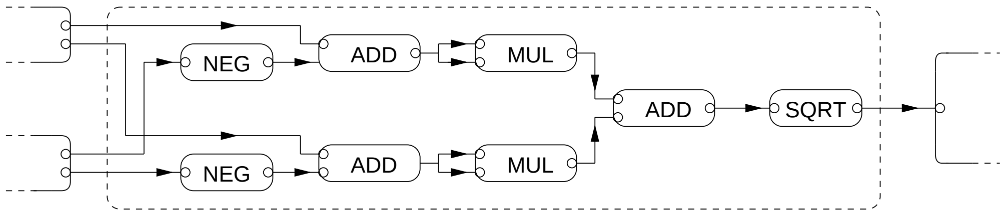

Figure 1: Subgraph of a data flow graph.

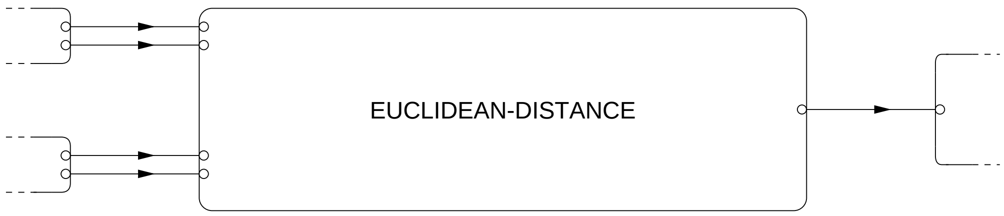

Figure 2: User-defined compound block.

In the case of software realizations, in contrast, abstraction through modular and hierarchical composition is not as simple [41, pp. 252–252]. The problem arises when efficiency matters—as it is rather usual with embedded computer systems—and assembly language is consequently a must. An assembly code segment is not context-free: for instance, the segment cannot use a given register as a temporary data storage if the surrounding code already happens to use the register so that it must not be overwritten. A possible solution attempt might be to employ functions with standard calling interfaces as program modules, but the extensive use of function calls would be very inefficient. Therefore, assembly language programmers seem to be in need of a more general notion of a program module.

In this report we propose a novel approach to the composition of machine (or assembly) code libraries. We have adopted several complementary ways to promote the proposal. After presenting some background in Sec. 2, in Sec. 3 we give a systematic description of our library-based code generation technique. Next, Sec. 4 introduces a demonstration prototype of the suggested code generation tool (moreover, the three appendices of the report deal with this prototype and its implementation). In Sec. 5 we provide a formal framework that should make the principles of our approach explicit and precise. The report ends with a brief identification of related research efforts, in Sec. 6, and a concise statement of our main conclusions, in Sec. 7. Additionally, we mention that for the up-to-date information about our contributions in this field you should consult the URL `ftp://saturn.hut.fi/pub/reflex/README` (in particular, you are told how to obtain a copy of the prototype tool and how to experiment with it).

2 Background and motivation

We are interested in developing programming tools and techniques in particular for embedded special-purpose processor code generation. On the whole, code generation must obviously take place not later than running the program. Still, this "constraint" allows several schemes distinct from the conventional *static* compile-time code generation. *Interpretation* means that the code for each source language statement is generated each time when the statement is executed; *dynamic* code generation means that program code is generated as the first step of the program execution, with the run-time parameter values already known (see [67], for example). At the other extreme, there is embedded processor *firmware* generation: code generation must take place in conjunction with the hardware design of the computing system, as the programs are physically closely tied to the hardware (e.g. stored in a read-only memory).

2.1 On embedded processor programming

The field of machine-level code generation is not homogeneous. The general-purpose *Reduced Instruction Set Computer* (RISC) processors are specifically designed to be programmed with optimizing high-level language compilers [80]: for instance, global register allocation [19] is straightforward due to the large uniform register file. But especially in the embedded real-time computing area, there are many special-purpose processor architectures that must often programmed in assembly language (which is not an easy task either). The unavailability of high-quality compilers stems from architectural difficulties, exceptionally high requirements on the output code quality, and limited compiler markets (the more specialized the processor is, the smaller the number of potential applications tends to be).

Embedded real-time computing systems [65, 41] are rapidly gaining ground. An example of an embedded system aimed at the consumer market is a mobile phone. Typically, a processor in such a system continuously runs a single program that must be able to respond very fast to some external signals. In spite of its elaborateness, this program may be relatively small. This is clearly different from the general-purpose computers on the office desk, which have to be able to switch between diverse application programs at the user's request.

Representative examples of the special-purpose processors we have in mind are the *digital signal processors* (DSPs) [66] designed for real-time number crunching. E. A. Lee states that they "are traditionally designed for performance, not extensive functionality or programmer convenience" [66, Part I, p. 4]. For the assembly language programmer, the main complication is the *fine granularity* of the instruction set (with VLIW-type instruction-level parallelism [36])—for execution speed, there is no intermediate microprogram level (this is a similarity to RISCs). For the compiler writer, the main problems are the non-RISC traits of the architecture: *unconventional functionality* with both limitations and exotic features, and *irregularity* and non-orthogonality. The architecture is thus strongly geared towards the intended application area. For instance, special-purpose registers (which need no explicit identification) are helpful when an architecture designer tries to pack into a single instruction word several operations that typically occur in conjunction with each other.

The core of the difficulties faced by the compiler is the case analysis problem described by W. A. Wulf in 1981 [103]: due to the architectural anomalies, selection of the optimum machine instructions and registers becomes troublesome. For a particular feature in the source program, there may be—instead of a single obvious implementation—only several alternative more or less dubious implementation candidates. Furthermore, individual implementation decisions are typically very sensitive to each other. For instance, some operation may "naturally" leave its result into a certain register, while the next operation using this result may expect its operand to be found in a different register. In practice, the compiler is seldom able to tackle a chain of such intertwined decisions in the best possible way. (In contrast, the code selection is usually a great deal easier for a human assembly language programmer, who rather quickly learns to "see" the best—or at least a very good—implementation alternative.)

In addition to the poor quality of compiled code, the are also more "managerial" issues that favor assembly language programming:

- In the case of a mass-produced article such as a mobile phone, increases in the non-recurring programming costs are usually acceptable [41, p. 9] when they result in a decrease in the recurring hardware costs.
- Special-purpose processor applications are often so specialized that there is little need to consider application program portability.
- It may be feasible to adopt a heterogenous multiprocessor system tailored to the particular application at hand: precisely the most time-critical tasks can then be assigned to a separate special-purpose processor programmed entirely in assembly language.
- One might suppose that the inevitable advances in integrated circuit technology and processor architecture design would soon more than compensate the performance penalty resulting from

> high-level language use. However, that is not the whole truth: as the processors get faster, software (or firmware) implementations become competitive in new application areas, but only if the full processing capacity can be exploited. In other words, there seems to exist a niche for assembly language programming, which is not easily occupied by compiler writers without some significant breakthroughs.

The above issues are reflected even in the embedded processor compiler technology [74]: compilers should absolutely be capable of powerful optimization. General-purpose processor compilers typically optimize for high execution speed, but with embedded processors, optimization for small code size may be even more important [71]: the smaller program memory is needed, the smaller are the recurring costs for each product sold. Finally, as the programming costs are non-recurring, optimization algorithms that consume exceptionally large amounts of time or space may become feasible [69, p. 26] (as a rule, the compilation does not take place on the target processor but on a more general and user-friendly platform).

2.2 Our goals

Compilation is often conceptually divided into the *analysis* and *synthesis* phases: for instance, lexical and syntax analysis belong to the former, whereas code generation belongs to the latter. Ideally, the analysis phase involves no decisions with run-time consequences: it neither loses any information about the input program nor makes any commitments concerning the output program. In contrast, the synthesis phase comprises all the machine-dependent decisions that affect the efficiency of the output program. This difference explains why the quality of compiler writing tools seems to be much better in the case of the analysis phase: Lex and Yacc are examples of proven scanner and parser generators, respectively [68].

Our present focus is on ensuring that a code generator program is always capable of producing output of sufficiently high quality. This is an issue especially with special-purpose processors—already the first Fortran compiler [8] launched in 1957 could produce sufficiently good code for a general-purpose processor, i.e. IBM 704 (since precisely ensuring this was crucial for the success of Fortran [9], the compiler featured even global program flow analysis). Accordingly, much of the contemporary code generation research concentrates on compiler retargetability and compilation speed rather than on further raising the level of output code efficiency. This latter task is typically relegated to optimization phases separate from the actual code generation; but, in practice, the less regular and orthogonal the processor architecture is, the less global optimizations can account for.

Instead of maximally automated machine-description-driven "high-end" code generation, we are presently aiming at a "low-end" code generation tool that would enable the compiler writer to fulfill his task in the first place, even when faced with exceptionally high requirements. Most of all, the tool should *assist* the designer by releasing him from tedious clerical work. Our specific goals are:

- The tool should be usable both as a code-generator writing system and as a stand-alone macro assembler.
- An experienced application programmer should himself be able to modify the code generation rules, and the quality of the output code should be a monotonously—and even rather steadily—increasing function of the time spent in such optimization.
- Retargeting the tool should be still be straightforward: even programmable *application-specific instruction set processors* are rapidly becoming more widely used [82].

We finally remark that, in principle, retargeting of a low-level tool of the suggested kind can be speeded up by combining it with some higher-level code generation tool in the manner sketched in Fig. 3. The low-level tool is shown to consist of a rule interpreter (only to make the figure simpler, we assume here that the low-level code generation rules are not subject to any preprocessing) and a rule base that may optionally be produced automatically by the higher-level tool. Most importantly, note that in the automated case the low-level rule base can still be improved by manual optimizations that are *not* lost even if the source program is subsequently modified (though they are lost whenever the target machine description has to be modified).

2.3 Requirement analysis

Our strategy is to strive for a machine-level code *abstraction formalism* with the following properties:

1. *Universality*: there should be no restrictions whatsoever on the code that can be generated.
2. *Transparency*: the designer should be able to "see" the structure of the output code effortlessly by viewing the abstract code.
3. *Modularity*: there should be support for reusable program components with well-defined interfaces and mutually independent implementations.

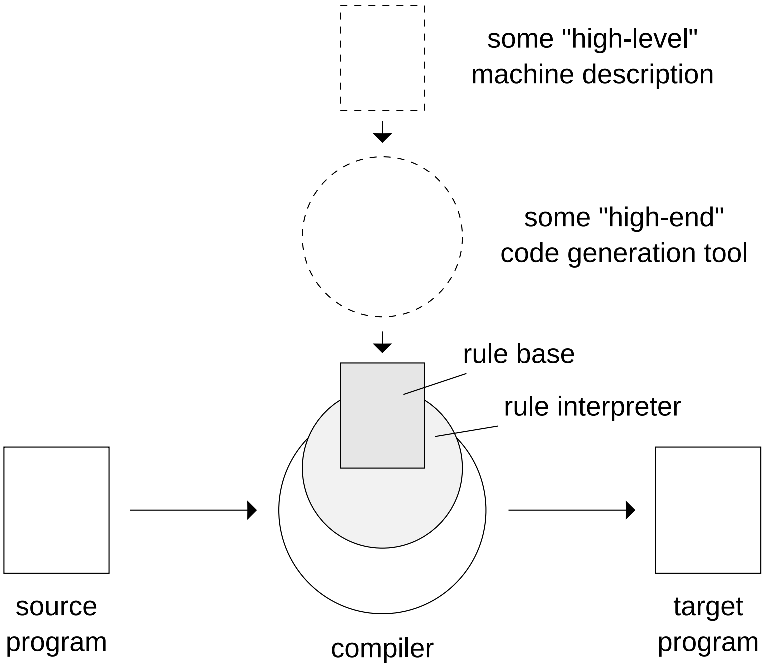

Figure 3: A scheme for faster retargeting.

4. *Hierarchy*: it should be possible to create new abstraction levels on the top of the already existing ones.

How do these properties help us to achieve the three specific goals set in Sec. 2.2? First, the main defects with existing code-generator writing systems and macro assemblers are the lack of universality and modularity, respectively: the compiler optimization level cannot be tuned arbitrarily high; and there is no protection mechanism that would prevent the macro expansion result from inadvertently corrupting the register contents. Second, assume that our code generation result is not good enough: hierarchy guides us in identifying the whereabouts of the defect; transparency helps us to recognize the reason for the defect; universality guarantees that we are able to remove the defect; and by modularity, the necessary modifications can be made locally. Third, suppose that we want to switch to a new processor: universality means that the formalism is indeed compatible with the new processor; hierarchy enables straightforward construction of processor-specific code libraries; modularity makes the libraries easily maintainable and safely reusable; and by improving readability, transparency encourages us to set up deeper library hierarchy.

We may further define *conformability* as the combination of universality and transparency, and *composability* correspondingly as the combination of modularity and hierarchy. Even without any abstraction formalism, one already has conformability, which may only be lost by a “poor” formalism; clearly, one usually adopts an abstraction formalism specifically in order to achieve composability. By conformability, the programmer is able to determine the output code structure as precisely as he wishes—in a way that is sensitive to as large a context as necessary. Composability, then, enables easy-to-use code libraries to be written. High-level languages such as C provide composability but not conformability; instead of conformability, they offer—more or less—machine-independence and thus portability of both programs and programming experience. Even if machine-independence is not among our present goals, by stretching the requirement of universality over different target hardware we still aim at retargetability.

To support our claim that the above four properties are independent of each other, we present four "three-property" examples:

- Conventional assembly language macros lack modularity. For instance, if some macro M_1 uses a register R as a temporary data storage, the programmer has to take care when calling M_1: the macro expander is not able to issue a warning that the value possibly already stored in R will be lost during the execution of M_1 [62, pp. 15–17]. Additionally, if M_1, in turn, calls another macro, say M_2, then the original caller of M_1 must take into consideration even similar restrictions that perhaps concern M_2.
- The MIPS assembly instruction set [32, 81] contains *synthetic instructions* that the assembler may expand into a couple of machine instructions. Sometimes the assembler needs a temporary register for storing intermediate results; modularity is still guaranteed since one of the hardware registers is reserved for the exclusive use of the assembler. But as there is only a single reserved register, it has not been feasible to allow hierarchical synthetic instruction definitions.
- Bliss [105, 104] is a *machine-oriented higher-level language* [96] designed around 1970 for PDP-10 system programming. As Bliss was intentionally tailored to exploit certain features of the PDP-10 instruction set architecture, it lacks universality.
- Using a high-level language such as C with optional *inline assembly* instructions (see [95, Ch. 13], for example), one can to write thoroughly optimized code. Still, the compilation result of the C portions of the program is not fully transparent.

In principle, modularity ensures that a single modification in the abstract code does not disrupt the global consistency of the output code. However, it is still possible that the automatic consistency preservation involves numerous changes in different parts of the output code. Uncontrolled, such "fluidity" obviously impedes transparency; controlled, it may even improve transparency.

More specifically, in addition to conformability and composability we would also like to have *plasticity* comprising the following two constituents (which may seem mutually conflicting—but wait for Sec. 3.4):

5. *Stability*: The effects of a modification in the abstract code should be localizable within a bounded segment of the output code.
6. *Propagativity*: If two coding decisions must be consistent with each other, the programmer should be able to make them at a single point in the abstract code, independently of the distance between them in the output code.

3 Our approach to code generation

Our proposed rule formalism can be seen as a *macro language*. This classification creates a common frame of reference: you are probably at least somewhat familiar with the concepts and terminology related to the use of macros. (Below we will use terms 'programmer' and 'macro writer' interchangeably—even if the programmer is often actually a compiler writer.) Still, while claiming many similarities to conventional macro expansion, we also want to stress the differences. Our macro expansion is *domain-sensitive* rather than *general-purpose*; by general-purpose macro expansion we refer to the pure text string substitution that is characteristic to the more traditional, domain-independent macro expanders.

The domain-sensitivity manifests itself in *domain-specific attributes* and *expansion-integrated domain-specific actions*. The macro writer is provided with a set of built-in domain-specific attributes, whose values he may examine and against which the conditional expansion may thus take place. For instance, he may ask whether there is any registers free to be overwritten at the point of the macro call. Similarly, but much less importantly, a TEX macro writer may ask for the height of the text accumulated on the current page and use this value as a condition for a "hard" page break. (TEX by D. E. Knuth [59] is a typesetting program whose proprietary input language features a flexible macro facility; TEX macro packages such as L. Lamport's LATEX [61] may be very elaborate systems.) The essential property of such domain-specific attributes is that their implementation is hidden from the user. The expansion-integrated domain-specific actions, in turn, are likewise embedded in the macro expansion facility itself. More specifically, the expander is preprogrammed to perform routine but (from a human viewpoint) laborious code-generation-related transformations—instead of plain text string substitution—in conjunction with the expansion of each macro call. Such transformations may involve storage allocation, for instance. (In contrast, the domain-specific actions of TEX are *conversion-integrated*, i.e. separate from the macro expansion proper: the TEX macro expansion can be seen as a preprocessing phase whose output is the input to the actual conversion phase. By a 'conversion' we mean a non-transparent language translation not based on explicit rules, as opposed to macro expansion.)

Because a macro expansion—even if the domain-sensitivity leads to enhanced context-sensitivity—is an inherently local operation, the proposed formalism facilitates local optimizations but cannot express global optimizations. Nevertheless, we shall claim that it is surprisingly easy to integrate supplementary global optimization routines with the proposed macro expansion.

3.1 Concept of a macro call environment

As we already noted in Sec. 2.3, conventional assembler macros lack modularity. This lack effectively prevents hierarchical macro definitions: since any hidden change in the state of the processor's registers is unsafe, hierarchy cannot promote abstraction through *information hiding* [79]. Our aim is to make macros into proper modules; from the viewpoint of the proposed macro expander, macro calls are not only string patterns but logical entities corresponding to computational operations.

Modularity requires that the *module implementations* are independent of each other. The *module interface* should, first, be carefully defined to cover all the possible intermodule dependencies and, second, be carefully respected when the modules are processed to produce output code. The advantage of modularity is that if some module implementation is modified, the output code remains globally consistent, even if numerous changes may spread all over it.

Our strategy is to manage the intermodule dependencies by extensive *conditionality*: the macro expansion takes place conditionally against the knowledge of the run-time context that is available already at expansion time. Typically, this knowledge contains information about the utilization of the processor's physical resources. Such knowledge constitutes the expansion-time *environment* of each macro call instance. The macro writer is presented with an abstract *display* through which he can examine the environment; the display hides both the internal program representation and the built-in algorithms (such as ones performing flow analysis) working on it.

The most important problems we now have to tackle include the following:

- What are the constituents of the expansion-time environment the should be utilizable in the conditional expansion? It seems doubtful that we could somehow guarantee that we are aware of all the relevant types of information. (See Secs. 3.2.3–3.2.5.)
- What kind of mechanisms should the abstract display offer for the programmer examining the environment? They should, of course, be easy-to-use and have clear-cut semantics. (See Sec. 3.3.3.)
- Should the programmer provide the code with some additional information to be processed by the macro expander only? In some cases the expander might perhaps need help in order to be able to make useful inferences about the environments. (See Sec. 3.2.2.)
- Should the expander be able to verify that the programmer does

respect the environment? Such automatic verification could probably make programming less error-prone. (See Sec. 3.2.2.)

- In general, the macro expander should only offer sophisticated but transparent decision-making tools to the programmer. But are there, after all, even such decisions that could be wholly relegated to the expansion mechanism? How "compiler-like" should the macro expander be? (See Sec. 3.3.2.)

3.2 Orthostatic and metastatic module interfaces

The macro expander must organize the knowledge that is, in an implicit form, contained by the expansion-time environment into the interface that explicitly steers the expansion of the particular macro call instance. (Even the domain-specific attributes are evaluated against this intercase, which is actually hidden from the programmer by a "second-level interface", i.e. the abstract display.) It is convenient to divide the macro call interface into two fractions: the pre-expansion-time *orthostatic* fraction becomes known before the strictly expansion-time *metastatic* fraction. More specifically, the orthostatic fraction becomes fixed at the time the macro definition containing the particular macro call *specimen* is written, whereas the metastatic fraction becomes fixed only at the time each particular macro call *instance* is ready to be expanded. A macro call specimen within some macro definition may be expanded multiple times (because of recursion, for example): the orthostatic interface remains the same each time but the metastatic interface is likely to vary. Thus, each orthostatic macro call specimen typically corresponds to several metastatic macro call instances.

We are not able to give an exhaustive definition for the ideal structure of either interface fraction. But first of all, the orthostatic interface should include the macro name and the number of macro arguments. The main components of the metastatic interface, then, are likely to be the following:

- How is the physical location of each argument of the macro call instance to be accessed? (See Sec. 3.2.3.)
- What is the available temporary storage, i.e. the set of free registers and memory locations? We say that a *storage cell* is *free* if there cannot be a data item that both has earlier been written into it and will later be read from it. (See Sec. 3.2.4.)
- What possibly useful knowledge does the expander already have about the run-time contents of the registers at the point of the macro call instance? (See Sec. 3.2.5.)

3.2.1 Orthostatic interface: basic components

The name of the macro called fixes the required number of macro arguments (however, certain macros—that is, macro names—perhaps allow some of the arguments to be vectors instead of scalars). Furthermore, *input arguments* are separated from *output arguments*, and each call specimen of a given macro must have the same number of input (or output) arguments. For example, a `sub` macro would be likely to require two input arguments and one output argument.

In addition to input and output arguments, there are also *exit arguments*. They are *code labels* that specify the possible locations to which the macro call may relinquish the run-time control. Similarly to input and output ones, the number of exit arguments is fixed. All these three fixed numbers are independent of each other: a control flow branch with two exit arguments as branch destinations may well also have output arguments. (If the number of exit arguments is zero, the macro is typically a non-terminating loop, which waits for an external interrupt.) It seems practical to allow macros to have an implicit extra exit argument that is always associated with the immediately following macro call specimen. This convention reflects the default sequential flow of control (see Sec. A.4.1 for a particular realization of the convention).

If the macro requires two or more exit arguments, there is typically at least as many distinct *exit points* inside the macro definition. In contrast, we require that each macro has only a single *entry point*; it seems natural to adopt this restriction to improve readability and transparency. (Note that, in a similar fashion, C functions have a single entry point but possibly many exit points. The difference is that there is only one return location for a given C function call, whereas our macro calls may have multiple return locations.)

3.2.2 Orthostatic interface: extension directions

The three macro argument lists should be complete: for example, for each storage cell that the macro may write, there should explicitly be an output argument. Thus, provided that the macro writer sticks with direct addressing, the macro expander can, by simple program flow analysis [49, 55], precisely determine the metastatic free storage cell set at each macro call instance.

It may even be reasonable to further extend the scope of the orthostatic interface. As we do not want to constrain the semantics of the macros, any additional requirements would still be essentially syntactic restrictions—like the completeness requirement on argument lists. What kind of benefits could such additional restrictions offer?

- They would enable the expander to build up more precise metastatic interfaces (in the same vein as the argument list completeness allows the precise free cell sets to be found in case of direct addressing): unfortunately, the expander often has to remain unsure of a fact whose truth or falsity the programmer is interested in.
- They would enable the expander to verify more accurately that the programmer really respects the environment (by the argument list completeness, the expander can check that the programmer does not overwrite any non-free cell at least by direct addressing). Nevertheless, full verification is admittedly beyond our reach. (In particular, because it seems impossible to keep track of the exact value of an address register, the expander has to allow indirect writes to locations known only to the programmer.)
- If carefully formulated, they would probably improve readability (as does the argument list completeness).

Some examples of additional constraints of the suggested kind are given below in conjunction with the discussion of the main components of the metastatic interface.

3.2.3 Metastatic interface: argument access

Local variables inside a macro definition are either *parameters* or *temporaries*. The orthostatic parameters represent the metastatic arguments of the macro currently being defined; the macro writer provides the lower-level macro calls inside the macro definition with orthostatic *argument designators*, which represent the metastatic arguments of these lower-level macros. Some local variable of the calling macro (see Sec. A.4.2 for the details of a particular scheme) is a possible argument designator. (By the way, you might notice that we have no need for a separate notion of a *run-time* argument. Actually, our metastatic 'argument' is often a specification of the run-time argument that is available already at the expansion time; typically, this specification fixes the location of the run-time argument. Similarly, our 'argument designator' is the specification that is available orthostatically, i.e. when the macro definition is scanned before expansion.)

For a moment, suppose that only direct addressing can be used. In this case, the metastatic interface must reveal the *storage class* and *intraclass identity* of each input or output argument. For instance, if the actual metastatic argument happens to be located in the memory, then the metastatic interface must fix the address of the particular memory location—obviously, the argument cannot otherwise be referred to in the resulting output code. Moreover, it is the metastatic interface that should determine that the argument resides in the memory instead of some register: the macro definitions become more general, and thus shorter, as this storage class information need not be fixed already in the orthostatic interface.

With indirect addressing a more elaborate scheme is needed. The metastatic interface must now specify the particular indirection type, i.e. the addressing mode, and the required "parameters" of this type, such as the index of the address register employed. Nevertheless, when an indirectly addressed argument of the macro currently being defined is only silently passed forward to a lower-level macro called, then the writer of the higher-level macro need not even be aware of the indirection. In this case, he may simple use the parameter as the argument designator of the lower-level call, whose orthostatic interface now leaves the addressing mode unspecified. But let us assume that the macro writer, instead, selects to change the addressing mode. In this latter case, he must typically first find out the old mode, and then specify the new mode as a part of the orthostatic interface of the lower-level call.

3.2.4 Metastatic interface: temporary storage

If direct addressing were exclusively used, the expander could determine the set of free storage cells by straightforward flow analysis. With indirect addressing, the situation is, of course, more complicated. But the more accurate *pointer analysis* (see [29, 100], for example) we are capable of, the larger safe estimate for the free cell set we can obtain. (Note, however, that indirect addressing typically applies to memory locations, whereas the register file constitutes the most useful temporary storage.)

For a start, we might still want to adopt a simple but safe heuristics based on the following observations:

- Because the expander performs the intraclass storage allocation for the macro temporaries (see Sec. 3.3.2), the programmer is, by default, ignorant of their exact addresses (i.e. intraclass identities).
- The programmer cannot even indirectly address such a storage cell whose address is fully unknown to him (i.e. so unknown that he has even no pointer for accessing the cell).

Accordingly, we propose the following pair of rules:

1. If the writer of a macro explicitly extracts the address of an indirectly addressable storage cell (by using some special 'address-

of' operator), then the expander considers that cell non-free at all lower-level macro calls (resulting from the call of the current macro) encountered during the expansion.

2. The writer of a macro is responsible for returning no knowledge about the addresses of its arguments back to the caller through the contents of the output arguments.

Finally, note that the smaller safe estimate of the possible destination set the expander can construct for each indirect jump, the more temporary storage may become available. In practice, this means that the orthostatic interface should specify the exact set (which is not a very strong requirement).

3.2.5 Metastatic interface: register states

Often it is extremely useful if the macro writer knows for sure the exact value of some (status) register at the point of the macro call (see the discussion of *mode control*—or *residual control*—in [4, 70]). For instance, suppose that the target processor architecture contains a status bit that determines whether addition overflows are saturated or not, and that the bit is manipulated by some dedicated instructions. Suppose further that we want to write a macro that performs a saturated addition, and that it is too costly to turn the overflow mode blindly off at each instance of the macro call, since there seems to be little or no need for non-saturated additions. What we now need is some device telling us the actual overflow mode at each (metastatic) call instance of the addition macro. More generally, we need a device that can record the information about the run-time register states that becomes available already at expansion time.

We suggest the introduction of special expansion-time variables that can be associated with individual registers to keep track of their run-time contents. These variables, which we call *fluxions*, are sensitive to the run-time control flow, and their range includes a special value denoting the ordinary "unknown" case (i.e. the case in which the associated register may have different contents at different execution times of the particular code segment).

There is a natural and fundamental restriction on the usage of fluxions—this restriction both in practice justifies their introduction and is presupposed by the formal model to be presented in Sec. 5. *No macro call expansion result is allowed to modify the value of a fluxion unless this modification is specified already as a part of the orthostatic interface of the macro call.* Moreover, the orthostatic interface typically even includes the new value (or at least a reference to it that is resolved by the metastatic interface).

3.3 Overview of the expansion mechanism

The recursive macro expansion process generates a tree structure whose root is the original macro call. A macro expansion tree with eight leaves is shown in Fig. 4. As this tree also contains seven non-leaf nodes (the root included), its generation has so far involved seven expansion steps, which have been applied to the individual macro calls now represented by these seven nodes. We do not know if this particular, informally specified tree can be further expanded; when a tree becomes complete, the leaf sequence of that time constitutes the expansion result.

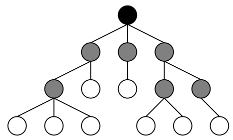

Figure 4: An expansion tree.

Next we present an overall description of the expansion mechanism. In particular, we are now interested in the possible realization of properties 1–4 listed in Sec. 2.3: universality, transparency, modularity, and hierarchy.

3.3.1 Macro semantics

Macros of any kind typically do not have any predefined semantics, contrary to the built-in elements of programming languages. The power of the proposed approach is in the generality resulting from this lack. For a simplified example, suppose that the target processor instruction set contains an *ADD* instruction that reads two registers and writes one register. Suppose also that the programmer uses the *ADD* instruction (i.e. the corresponding system macro) inside the definition of some macro *AVERAGE*. When the macro expander, then, tries to expand some call of *AVERAGE*, all that it has to do concerning the *ADD* instance is to select three appropriate registers (in particular, it must take care that the data possibly already present in the output register is not prematurely overwritten; see Sec. 3.3.2 for the main principles of storage allocation). Thus, it can ignore a great deal of information relevant to any high-level language compiler. Most importantly, it need not even know whether *ADD* actually performs an addition or something wholly different.

In all, the lack of predefined semantics offers the following advantages:

- The macro expander proper can be made a relatively small program. It is the rule base consisting of macro definitions that drives the expansion.

- Operations not needed in the chosen application need not be implemented. Still, if additional functionality is later required, new macros may then be added to the rule base.
- Universality is strongly supported: with suitable macro definitions, the tool can be tailored to fully exploit the particulars of the chosen target processor architecture.

To further promote universality, we accompany the lack of predefined semantics with a lack of structural restrictions: any macro call segment with a single entry point can be defined as a macro (for the singularity of the entry point, see Sec. 3.2.1). The base of the macro hierarchy is constituted by *atomic* macros, which are actually not proper 'macros' but typically placeholders for individual machine instructions. We say that the macros at the upper hierarchy levels are *composite* ones; a composite macro may call any atomic or composite macro—even (both direct and indirect) recursion is allowed. (Of course, the macro expansion process stops when all the leafs of the expansion tree are calls of atomic macros.)

Finally, what kind of constraints do we have to set on the target processor architecture? We believe that basically all sequential uniprocessors are covered by our approach (for *instruction-level parallelism* [36] we provide no explicit support). Nevertheless, we still stress that there is no support for any standard machine-independent interfaces within the macro hierarchy: in addition to the implementation of the atomic macros, even their names and semantics are intended to be fully processor-dependent.

3.3.2 Storage allocation

The expansion-integrated domain-specific actions are likely to be more clerical than intelligent. Most importantly, such built-in actions must be transparent: the programmer is not likely to devote himself to ultimate refinement of the conditional macro expansion if some transformation to follow may have seemingly unpredictable consequences.

Some macros employ temporary variables. This means that there must be some data storage free at the point of the macro call. But how should the free data storage be found and then allocated for the macro temporaries? (We completely ignore the memory allocation for the program code resulting from the macro expansion; we simply assume that there is always enough space for the code.) The suggested storage allocation scheme reflects the structure of the user-defined macro hierarchy: it is assumed that when a macro is (metastatically) called, the storage for the macro parameters has already been allocated; a part of the storage allocation for the possible macro temporaries, instead, is carried out as our main expansion-integrated domain-specific action. Since flow (and pointer) analysis is involved, this action is not a trivial one.

We assume that the target architecture provides a set of distinct *storage classes*, and that the *storage cells* in each class can be used fully interchangeably (a representative example of a storage class would thus be the set of memory locations, or the set of general-purpose registers). The division of labor is then as follows. The macro writer chooses the optimum storage class for each temporary, and the macro expander metastatically tries to find some free member of the chosen class. Thus, the programmer completely specifies the *interclass* storage allocation, whereas the expander autonomously performs the *intraclass* allocation.

The above sketched idealized scheme does not fully match the real processor architectures. Because simply specifying the storage class is, not always sufficient, in practice the macro writer also has to possess more elaborate allocation requests. In all, the most obvious bottlenecks of the basic scheme above include the following:

- In particular with RISC architectures featuring a large homogeneous register set, global register allocation methods are clearly superior to local ones (but see Sec. 3.3.4 for our proposed remedy).
- It should be possible to define two storage classes to be physically overlapping. For instance, there may be double precision registers comprising two normal precision registers each.
- There are often storage cells which are not functionally independent of each other. For instance, it might be so that only one of the two registers containing the operands of an addition operation is explicitly named in the instruction word, whereas the other operand is implicitly assumed to be found in the other member of the particular odd-even register pair.
- Especially in case of indirect addressing, the programmer should be able to specify that two variables that are used together should also be located near each other. This is because small address register modifications are often very cheap (see [71] for a further discussion).

3.3.3 Conditionality

Extensive macro hierarchy means smaller scopes for the programmer, who should be able to manage each scope—transparently, in a modular fashion, and without losing universality—by a versatile *conditional expansion* facility. Indeed, each (composite) macro definition may consist of several alternative implementation *versions* guarded by a condition that is metastatically evaluated by a powerful *expression interpreter*.

In 1960, M. D. McIlroy [75] stated that "any compiler should include within it an interpreter for its source language or somewhat equivalent". As opposed to conventional high-level languages, our macro system offers no predefined run-time operators (all operations supported by the target must be explicitly declared as atomic macros), but it does offer a powerful expansion-time expression interpreter that provides the usual arithmetic operations as predefined primitives. The most important feature of the interpreter is, however, a set of domain-specific attributes that constitute the abstract display; through this display, the programmer can examine the view that the macro interface enforced by the expander gives of the expansion-time environment. The idea behind the abstractness of the display is that the domain-specific attributes should implement transparent *information hiding*. For example, there might be an attribute meaning 'is there any free cell of storage class R available?'—thus the flow analysis necessary would be wholly hidden from the programmer.

Even if the guarding condition, or *version-specific test*, is fulfilled, the version may still have to be rejected. In all, the possible reasons for rejecting a version are the following:

- The version-specific test is not successful.
- The version writer happens to violate the environment of the particular macro call instance in a way that the expander is able to detect. For example, if the version writer explicitly intends to overwrite the input argument register, the expander should find out whether the environment expects that the value remains intact. (When the detectable violations are documented, the programmer may shorten the version-specific test by relying on them as sentinels.)
- The expander cannot perform the required built-in transformations (i.e. expansion-integrated domain-specific actions) without violating the environment. For example, temporary storage allocation is impossible if there is not enough free storage.
- The *macro-specific tests* of the lower-level macro calls in the version do not succeed (even if the macro expander should try to take these tests into account when carrying out the built-in transformations). Introducing a macro-specific test for a given macro may make other macro definitions shorter and more easily main-

> tainable: such a test can be seen as an implicit component of the version-specific test of each such macro version whose code contains a call of the given macro. (See Sec. 4.3 for programming examples; in contrast, the formal model to be presented in Sec. 5, for simplicity, ignores macro-specific tests.)

Within a macro definition, the alternative versions are listed in the order of decreasing priority—the programmer should thus place the most favored versions first. We believe that the expansion-time control flow should, for transparency, be most simple-minded: once the expander finds a version that matches the above criteria, it is irreversibly selected. But if no acceptable version is found, the whole expansion process originating from the root of the particular expansion tree fails. Thus, there is no backtracking: if the expansion of some lower-level macro call in the code of an accepted version fails, then this failure is a fatal one. As only the first match counts (in principle, costs of some kind could perhaps be employed for selecting between multiple matches), there is also a potential pitfall: the programmer is tempted to shorten the version-specific tests by adapting them to the particular order in which the versions are listed.

3.3.4 Interplay with global optimizations

Macro expansion suits especially well to local optimization but rather poorly to global optimization, which involves adjustment of a net of long-distance dependencies spanning all over the code. Nevertheless, we suggest that it is possible to compensate this inherent locality by combining add-on global optimization routines with the macro expansion.

To make room for global optimizations, we split the macro library into several "horizontal" slices; consequently, the expansion tree also divides into horizontal layers corresponding to the library slices. (Such a slicing is encouraged by an important property of the macro expansion to be discussed in Sec. 3.4.3.) The macro expansion is thus driven by an ordered pack of macro libraries that are loaded into the expander one at a time in a strictly sequential fashion. Each time the macro expander becomes ready with a library slice (that is, when all the remaining macro calls are atomic with respect to that slice), it steps aside for a moment, as some slice-specific global optimization routine may temporarily take over; this scheme is depicted in Fig. 5 (see Sec. 5.5 for a more precise description). What the optimizations are like is of no concern to the expander, which in due time recovers the control and starts afresh with a new library slice and a possibly modified sequence of macro calls to be further expanded.

Prominent add-on optimization candidates include, for example, a global en-

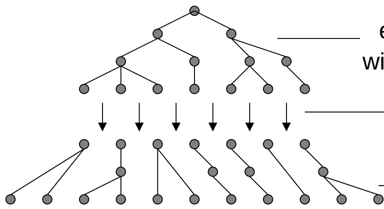

Figure 5: Informal view of a global optimization.

hancement of the register allocation as an intermediate processing phase and *code compaction* [35, 54] as a postprocessing phase.

3.4 Interaction between macro calls

We have noted that macro expansion suits better to local than global optimization. Still, here we explain what kind of communication is possible across the boundaries of neighboring macros, and whether such communication sets any constraints on the expansion order of individual macro calls. This communication scheme, which is a distinctive feature of our approach, promotes the remaining properties 5 and 6 listed in Sec. 2.3: stability and propagativity.

Below, we first provide some background. In the case of general-purpose macro expansion, intermacro communication requires that one introduces global variables and fixes a strict depth-first left-to-right macro expansion order (to simulate the default execution order of machine instructions). We feel that these two closely intertwined conventions are strong deficiencies: global variables are subject to hidden side effects which make the macro libraries difficult to maintain; the left-to-right requirement prevents parallel macro expansion.

3.4.1 Some contrast: on general-purpose macro expansion

With general-purpose macro expanders [17, 16, 20], conditional expansion is domain-independent. It takes place against the expansion-time values—which may be arbitrary text strings—associated with certain identifier tokens. The identifiers involved are divided into *parameters* and *global variables*: the parameters inherit their values from the corresponding arguments of each call instance of the macro; the value of any global variable can be changed with an explicit redefinition directive. As the source text is processed strictly sequentially, the effective definition of each global variable is always unique, i.e. the most recent one encountered.

P. J. Brown [16, Ch. 1.7] demonstrates how the intermacro communication by global variables can be utilized in code optimization. Suppose that we have written a two-argument macro that copies the contents of one memory location into another on our single-accumulator machine. For instance, the macro call sequence below on the left might produce the result shown on the right.

```
Move M1 M2                                              load M1
                                                        store M2
Move M2 M3                                              load M2
                                                        store M3
```

The above result is, however, not an optimum one: the second load from memory is readily seen to be redundant. This deficiency can be eliminated by conditional expansion: we can introduce a global variable, say `$Acc`, for representing our knowledge of the run-time contents of the accumulator. Now, if the first `Move` instance set `$Acc` to value '`M2`', then the second `Move` instance could examine the value of `$Acc` and consequently drop out the redundant `load` instruction. (This suggestion is not a very general one yet: it would not work if the second `Move` instance were replaced with an equivalent macro call reading as '`Move M1 M3`'.) In a similar vein, an instruction clearing the accumulator could be optimized off if it occurred immediately after a 'branch-on-nonzero' instruction.

There are, of course, inherent problems with any simple-minded technique of the above kind. The strictly sequential left-to-right macro call processing order effectively enables the macros to "look back", but they cannot "look ahead" to find out, say, in which register the next operation would like its operands to be located. Moreover, the above technique can be applied only to basic blocks, i.e. straight-line code segments, whereas control flow branches cannot be coped with. These limitations—which the present study attacks—were expressed by Brown in 1974 [16, p. 62]: "Macro processors simply do not have the facilities to look at the dynamic behaviour of a program and optimize on the basis of this."

3.4.2 Context-sensitivity

Our approach excludes globally visible expansion-time variables. Nevertheless, not all information passed to a macro call needs to be explicitly formulated as macro arguments. This is because the macro calls are *context-sensitive*: for example, they are effectively aware of the free storage cell sets. Such implicit intermacro communication makes the macro definitions more readable and maintainable, as the programmer need not provide the macro calls with exhaustively long and tightly interdependent argument lists consisting of mainly redundant information. Clearly, the context-sensitivity improves propagativity; still, the horizontal propagation is possible only within macro calls that are issued within the same macro (or rather, version) definition (see Sec. 5 for a precise description). This scoping restriction implies that the programmer should make the most important decisions on the upper levels of the macro hierarchy to enable them to propagate farther.

The context-sensitivity rests on the assumption that the expander builds up the metastatic interfaces. This may seem like a lot of work, but fortunately, the determination of the metastatic interface can be done lazily: the expander calculates only those components of the interface that are explicitly examined by the programmer.

3.4.3 Order-independence

The main property of the model to be presented in Sec. 5 is *order-independence*: the final macro call expansion result is independent on the expansion order of the individual macro calls. This property, which is depicted in Fig. 6, stems from the fact that the environment of a macro call never changes due to the expansion of other macro calls: by definition (see Sec. 5.1), in all the four expansion trees of Fig. 6 the environment of node N remains the same. The order-independence means that there is neither the left-to-right bias nor the basic-block restriction of the general-purpose macro expanders. Effectively, the proposed macro expansion is *functional*: there is no hidden side effects.

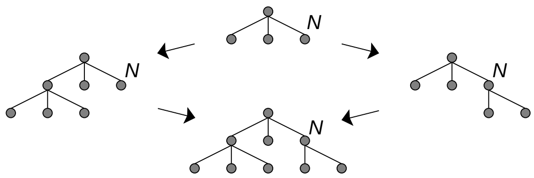

Figure 6: Order-independence.

The order-independence has several advantageous consequences:

- Transparency of the macro expansion is enhanced.
- Macro calls can be expanded in parallel [85, Ch. 24], which enables efficient implementations.
- As the expansion-time environment of a macro call never changes, laziness in determining the components of the metastatic interface is justified.

- It is straightforward to split an existing macro library into two or more slices and introduce intermediate global optimizations. The reason is that as the expansion order can freely be selected in any way that seems desirable, the expander might just as well process the macro calls in such a layer-by-layer basis whose layers exactly correspond to the library slices.
- Stability of the macro expansion is enhanced: if some macro definition is modified, only such code that results (possibly indirectly) from a call of the modified macro may change. Thus, outside the modified macro definition itself, the changes can propagate only downwards—not left, right, or upwards. This means that if the programmer makes the most important decisions on the upper levels of the macro hierarchy, as suggested in Sec. 3.4.2, their consequences are not affected each time a minor decision on a lower level is reconsidered.

We still want to illustrate the stability claimed above. How a code modification may propagate in an expansion tree is depicted in Fig. 7. Suppose that inside the definition of macro *M0*, the argument list of the call of *M2* is modified. By context-sensitivity, this modification propagates across the macro calls that are siblings of the *M2* call, so that the calls of *M1* and *M3* may be affected. Furthermore, any descendant of an affected macro call may also be affected. But most importantly, no white-colored node in the tree of Fig. 7 is influenced by the modification.

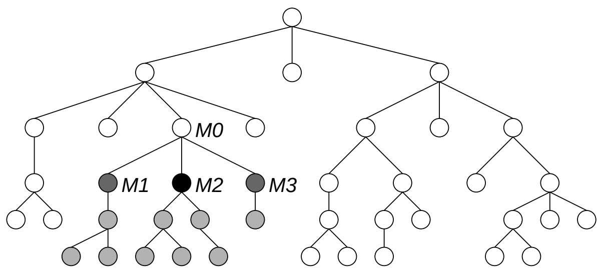

Figure 7: Propagation of a modification: our approach.

In contrast, let us examine the effects of a similar change in the case of general-purpose macro expansion; the situation is depicted in Fig. 8. Again, we assume that inside the definition of *M0*, the argument list of the *M2* call is modified. Now the general-purpose macro expander sticks to the left-to-right depth-first expansion order, and only such macro calls (which are again white-colored in Fig. 7) that the expander processes *before* the *M2* call are guaranteed to remain unaffected by the modification.

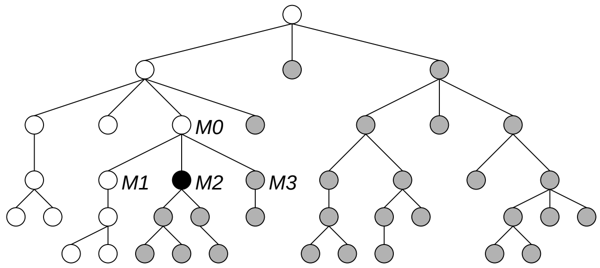

Figure 8: Propagation of a modification: general-purpose macro expansion.

4 A prototype implementation

We have implemented a simple demonstration prototype of the proposed macro expander; the ReFlEx 1.0 program is freely available for non-commercial purposes (see Sec. 1). This prototype was introduced in [64], a more detailed reference is found in Appendices A and B, and the complete reference is [63] (in addition to providing a full user documentation, that report is precise enough for an interested reader to build a re-implementation with the same functionality). The main content of the present section is a tutorial example (Sec. 4.3) of the use of ReFlEx 1.0; more examples are given in Appendix C and in [63].

4.1 Objectives

Our main objectives in the design of ReFlEx 1.0 were:

- To demonstrate the basic ideas of our approach.
- Specifically, to demonstrate that these basic ideas are sound.

Thus, the following were *not* among our goals:

- Extensive functionality: diversity of details might hide our basic ideas.
- Ultimate efficiency of the expander implementation (which would not affect the quality of the expansion results): the example cases processed by ReFlEx 1.0 will probably be rather small.

Consecutively, we were in a rather short time able to complete and document prototype implementation.

4.2 Limitations

The rudimentary ReFlEx 1.0 prototype system supports only unrealistically simple processor architectures; still, it is fully retargetable within its restricted class of target architectures. The one elaborate feature of ReFlEx 1.0 is that by flow analysis it can at any point of code precisely determine the set of the data storage cells that are guaranteed to be free.

The main restrictions on the target architecture include the following:

- It is not possible to declare data vectors—only scalar variables can be used.

- Indirect addressing is not supported—the only possible addressing modes are direct and immediate. (Of course, this restriction dramatically simplifies the building of the metastatic interface, especially the construction of the free storage cell set—see Sec. 3.2.4.)
- Any two cells in different data storage classes must be physically distinct (see Sec. 3.3.2).
- Only the storage class of each macro temporary can be specified by the macro programmer (see Sec. 3.3.2).
- There is no explicit support for intermediate global optimizations (see Sec. 3.3.4).
- There is no explicit support for instruction-level parallelism (for instance, VLIW-type parallelism might be supported by a code compaction routine, which is an example of a prominent global optimization).
- Fluxions (see Sec. 3.2.5) are not provided.

4.3 A tutorial example

ReFlEx 1.0 reads its macro definitions from an ASCII-formatted *rule file* produced by the user; with these macro definitions, it is then able to generate code for a given macro call. In a step-by-step fashion, we will here write a simple rule file and finally run some elementary code generation examples. The main parts of the rule file are shown in Fig. 9.

```
HEADER {     ...     }
STORAGE {     ...     }
SYSTEM {     ...     }
UTILITY {     ...     }
```

Figure 9: Overall structure of the rule file.

The `HEADER` part contains auxiliary definitions for the expansion-time expression interpreter. In the `STORAGE` part one specifies the available run-time data storage, i.e. the CPU registers and the data memory. The atomic and composite macros are described in the `SYSTEM` and `UTILITY` parts, respectively. An atomic macro is simply a placeholder for a machine instruction of the target, while a composite macro is a proper macro, i.e. a compound consisting of calls of another—atomic or composite—macros.

4.3.1 Expansion-time expression interpreter

In the HEADER part you may define new expressions for the built-in expansion-time expression interpreter. For instance, here is the recursive definition for the factorial function:

```
Fact(a) = _If(_Lt(a,2), 1, _Mul(a,Fact(_Add(a,-1))));
```

The expressions recognized by the interpreter are called *forms*. Here the forms _If, _Lt, _Mul, and _Add are *primitive*, that is, predefined. Not surprisingly, _Lt, _Mul, and _Add are 'less-than', multiplication, and addition, respectively. As for _If, if its first argument is nonzero, it returns the value of the second argument; otherwise, it returns the value of the third argument. This special primitive form supports lazy evaluation (in the case of Fact, for example): never are all three of its arguments evaluated.

Then, Fig. 10 introduces some more user-defined *compound* forms:

Not	Is the argument zero?
And	Are the two arguments both nonzero?
Or	Is at least one of the two arguments nonzero?
Int	Is the argument an integer?
Lt	Is the first argument an integer that is less than the second one?
Gte	Is the first argument an integer that is greater than or equal to the second one?
Eq	Are the arguments equal integers?

```
Not(a) = _Nand(a,a);
And(a,b) = _If(a, _If(b,1,0), 0);
Or(a,b) = _If(a, 1, _If(b,1,0));
Int(a) = And(Not(?(a)), Not(&(a)));
Lt(a,b) = And(_Lt(a,b), And(Int(a),Int(b)));
Gte(a,b) = And(Not(Lt(a,b)), And(Int(a),Int(b)));
Eq(a,b) = And(Gte(a,b), Gte(b,a));
```

Figure 10: Compound form definitions.

Most of the primitive forms mentioned above are *autonomous*; the exceptions are ?(a) and &(a), which are *diagnostic*. Diagnostic primitives can be used for extracting information about the expansion-time environment. We give some examples (if a diagnostic primitive takes arguments, it accepts any number of arguments, and there is an implicit conjunction between the arguments):

`?(x,y)` Do `x` and `y` represent *cells*, i.e. data storage elements?

`?M(x)` Does cell `x` belong to class `M`?

`?R(x)` Does cell `x` belong to class `R`?

`#R()` How many of the cells in class `R` are currently free?

`&(l0,l1)` Are labels `l0` and `l1` equivalent, i.e. do they refer to the same code location?

4.3.2 Run-time data storage

Our hypothetical target processor architecture is utterly simplified. As shown in Fig. 11, there are 1024 memory locations `M`[0]–`M`[1023], four auxiliary registers `R`[0]–`R`[3], and a single accumulator `A`. Thus, there are three distinct storage classes. ReFlEx assumes that the cells in each single storage class can be used fully interchangeably. (Note that we were able to refer to the storage classes already in the compound form definitions.)

```
STORAGE {
  M[1024];
  R[4];
  A[1];
}
```

Figure 11: Declaration of data storage cells.

ReFlEx itself need not know the cell lengths (which may well be different for different classes) in terms of bit positions. On the other hand, in the case of a real processor the user should absolutely be familiar with this information.

4.3.3 Machine instruction set

Next, in Fig. 12 we declare the atomic macros, which represent the machine instructions of the target. What we define is actually only the interface seen by composite macro writers. ReFlEx cannot convert atomic macro calls into real machine instructions; what it is able to do, is to convert composite macro calls into atomic macro call sequences. It seems clear that this latter task is the interesting one, while the former task should be a routine matter and therefore relegated to a simple ReFlEx-compatible assembler.

So there are, in all, ten atomic macros, among which there are one unconditional branch (`JUMP`) and three conditional branches (`BRANCH`). Most of the atomic macros are provided with a *macro-specific test* (`TEST`) that constrains

```
SYSTEM {
  set(c > r) { TEST And(?R(r), And(Gte(c,-1024),Lt(c,1024))); }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) { TEST And(Or(?A(s),?R(s)), Or(?A(d),?R(d))); }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}
```

Figure 12: Declaration of atomic macros.

their use. By providing the `load` atomic macro definition, for instance, with such a test, the programmer makes macro definitions shorter and more easily maintainable: otherwise, every single call instance of `load` should be guarded by a *version-specific test* that would check whether the arguments really are a memory location and an auxiliary register.

ReFlEx does not know the semantics of the atomic macros; nevertheless, this semantics must, of course, be known to the composite macro writer. With the information seen in Fig. 12, even the following brief description should be fairly comprehensive for a prospective macro writer:

- `set` "reads" a signed 11-bit integer and writes it into one of the auxiliary registers (notice that form `?R` checks whether its argument belongs to storage class `R`). Thus, `set` is for immediate addressing, as the value of the integer is fixed at expansion time.
- `load` copies the contents of a memory location into an auxiliary register, and `store` performs the opposite data transfer.
- `move` can move data between two storage cells, provided that each one of the cells is either the accumulator or an auxiliary register.
- `add` adds the contents of one of the auxiliary registers into the accumulator. Thus, it reads and writes the accumulator, and additionally reads an auxiliary register. Similarly, `sub` performs a corresponding subtraction.
- `goto` jumps to the specified code label.
- `eq`, `gt`, and `lt` branch to the specified label if the contents of the accumulator are, respectively, equal to zero, positive, or negative. Thus they all read the accumulator.

From this rather restricted atomic macro set, one may infer that the (hypothetical) target processor instruction set is similarly restricted. Note in particular that the accumulator cannot be loaded directly from memory.

4.3.4 Higher-level macros

Now we are ready to define composite macros, without which the rule file would not be of any use. Our first composite macro, `my_null` shown in Fig. 13, is simple: it does not do anything. Still, it is a useful macro, as you will see later. (Actually, the essential feature of `my_null` is the requirement that the input and output arguments must be the same cell.)

```
my_null(x > x) {
  null: { }
}
```

Figure 13: Definition of the `my_null` composite macro.

The `my_move` macro in Fig. 14, then, implements a general data transfer not subject to any storage class restrictions, contrary to the atomic macros declared above. The macro definition consists of six alternative *versions*, each with a distinct name. The versions are listed in the order of decreasing priority, i.e. in the order in which ReFlEx should try to apply them to each `my_move` call.

```
my_move(s > d) {
  same: { my_null(s > d); }
  as_set: { set(s > d); }
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  temp_needed: TEST And(Not(?R(s)), Not(?R(d))); USE R[r];
    { my_move(s > r); my_move(r > d); }
}
```

Figure 14: Definition of the `my_move` composite macro.

Note that the `as_load` version, for instance, can be accepted only if the source of the data transfer is a memory location and the destination is an auxiliary register, because the macro-specific test of `load` in Fig. 12 rejects calls of other kinds. In Fig. 14, in contrast, the single explicit test is version-specific.

Let us look closely at each one of the `my_move` versions:

- `same` guarantees that the expansion result is an empty code sequence when the source and the destination are the same cell. The empty my_null call actually serves two purposes: it verifies that parameters s and d do represent the same cell, and it "writes" the output parameter d. (To demonstrate its flow analysis capability, ReFlEx 1.0 checks that output-only parameters are not left unwritten.)

- as_set, as_load, as_store, and as_move may only be converted into the respective corresponding atomic macros.
- temp_needed requires a free auxiliary register. This register is used as an intermediate storage when, for instance, the contents of a memory location are to be transferred into the accumulator. The version is recursive, but the recursion depth can be seen to be at most one.

Our third composite macro, my_mswap shown in Fig. 15, interchanges the contents of two memory locations. Unlike our first two composite macros, my_mswap has a macro-specific test (i.e. one that verifies the storage classes).

```
my_mswap(m,n > n,m) {
  TEST ?M(m,n);
  two_aux_free: TEST Gte(#R(),2); USE R[r];
    { my_move(m > r); my_move(n > m); my_move(r > n); }
  acc_and_aux_free: USE A[a];
    { my_move(m > a); my_move(n > m); my_move(a > n); }
}
```

Figure 15: Definition of the my_mswap composite macro.

To perform the swap, my_mswap requires two cells of free storage, because direct transfers between memory locations are not supported by the atomic macro set. At least one of these two cells must be an auxiliary register, while the other one may alternatively be the accumulator. Accordingly, form #R, seen in the my_mswap definition, returns the number of free cells in storage class R. (The version-specific test of two_aux_free is necessary, because in the version code only one auxiliary register, i.e. r, is written, but the middle one of the lower-level my_move calls certainly needs another one. Without the test, two_aux_free might be selected even in such a case that the acc_and_aux_free version might be the only fully expandable one of these two (such a mistake would be fatal as backtracking is not supported).

We can also define macros that are (conditional or unconditional) branches: the data flow analysis performed by ReFlEx 1.0 is able to cope with control flow branches. Our final composite macro shown in Fig. 16 is a conditional 'branch-if-positive' operation without storage class restrictions.

```
BRANCH my_gt(x) [l] {
  next: TEST &(l,NEXT); { }
  const0: TEST Gte(x,1); { JUMP goto() [l]; }
  const: TEST Int(x); { }
  acc: TEST ?A(x); { BRANCH gt(x) [l]; }
  default: USE A[a]; { my_move(x > a); BRANCH gt(a) [l]; }
}
```

Figure 16: Definition of the `my_zero` composite macro.

4.3.5 Generating code for macro calls

Now we put our macro definitions into use and produce some code for our hypothetical target processor. We start with a single macro call: ReFlEx 1.0 expects that the expansion source is represented as one top-level macro call.

Suppose that ReFlEx has been installed on our workstation and our rule file is called `simple.m`. We start ReFlEx by typing the following command (the option `-t` provides us with some interesting optional output, i.e. the full-blown intermediate expansion tree):

```
reflex -t simple.m
```

If ReFlEx starts successfully, we may then type the following expansion source as a response to the ReFlEx prompt, i.e. '>':

```
> my_mswap(M[8],M[6] > M[6],M[8]); {R[1],R[3]}
```

This means that we want to exchange the contents of data memory locations M[8] and M[6]. Additionally, we specify the auxiliary registers R[1] and R[3] are free at the macro call (an explicit specification like this is needed and allowed only in the expansion source). The freedom at a given macro call simply means that the realization of the macro call may write the particular cell.

The expansion result is shown in Fig. 17. The intermediate output at the top reveals the expansion-time tree structure, which is flattened in the final output at the bottom. (For convenience, our text-formatted expansion trees grow to the right, whereas graph-formatted ones grow downwards—see Fig. 4 on page 17. Note also that the actual output syntax of ReFlEx 1.0 is unfortunately somewhat more cryptic than suggested here.)

We would like to stress the crucial point of this expansion result, which demonstrates the advantage of the program flow analysis capability. When ReFlEx tries to expand the `my_move(M[6]>M[8])` call at the first level below the initial `my_mswap` call, it recognizes that the originally free auxiliary

```
my_mswap(M[8],M[6] > M[6],M[8])  {R[1],R[3]}
    my_move(M[8] > R[3])
        load(M[8] > R[3])
    my_move(M[6] > M[8])
        my_move(M[6] > R[1])
            load(M[6] > R[1])
        my_move(R[1] > M[8])
            store(R[1] > M[8])
    my_move(R[3] > M[6])
        store(R[3] > M[6])

load(M[8] > R[3])
load(M[6] > R[1])
store(R[1] > M[8])
store(R[3] > M[6])
```

Figure 17: Expansion result of a `my_mswap` macro call.

```
my_mswap(M[8],M[6] > M[6],M[8])  {A,R[3]}
    my_move(M[8] > A)
        my_move(M[8] > R[3])
            load(M[8] > R[3])
        my_move(R[3] > A)
            move(R[3] > A)
    my_move(M[6] > M[8])
        my_move(M[6] > R[3])
            load(M[6] > R[3])
        my_move(R[3] > M[8])
            store(R[3] > M[8])
    my_move(A > M[6])
        my_move(A > R[3])
            move(A > R[3])
        my_move(R[3] > M[6])
            store(R[3] > M[6])

load(M[8] > R[3])
move(R[3] > A)
load(M[6] > R[3])
store(R[3] > M[8])
move(A > R[3])
store(R[3] > M[6])
```

Figure 18: Expansion result of another `my_mswap` macro call.

register R[3] is not free any more. Thus, R[1] is the only possibility for a temporary storage at the next-lower level. In other words, if there were only one originally free auxiliary register, the expansion would fail.

Finally, suppose that in the previous example we had had the accumulator (as the size of class A is 1, 'A' may be used for 'A[0]') and one auxiliary register free, instead of two auxiliary registers. Then we would have got the output shown in Fig. 18.

5 A formal model

In this section we present a formal model for domain-sensitive macro expansion. This model is a theory consisting of *specifications*, which identify our basic building blocks, *definitions*, and *propositions* (or theorems). We have a great number of specifications, as we want to make our vocabulary explicit; furthermore, to strive for wide applicability, we have tried to make the model open-ended (the specifications may be regarded as concerning external interfaces). Our few propositions, in turn, are neither surprising nor established by particularly ingenious proofs. On the contrary, we simply selected in advance the set of the fundamental properties that the model was then rather mechanically constructed to possess.

Our intended domain is machine-level synthesis of computer programs, but the model is domain-independent, that is, it could have domain-specific instantiations even in other domains. Understandably, the model is more abstract than our presentation in previous sections. For example, the data flow (but not the control flow) of a computer program is now fully ignored. To make the model more easily approachable, we have adopted a small deviation from the expansion mechanism described in Secs. 3 and 4: *macro-specific tests* are excluded here (see Sec. 3.3.3).

5.1 Program representation

In this section we give a structural description of an expansion tree. This description will be semantically refined in Sec. 5.3; tree restructuring, i.e. the actual expansion process, will be discussed in Secs. 5.4 and 5.5.

Specification 5.1
(a) *There is a set M of **macro names**, a set Y of **macro call cores**, and a function $\mu_Y : Y \to M$.*
(b) *There is a countably infinite set B of **labels**.*
(c) *There is a set C of **macro calls**. Each macro call c is a triple of the form $\langle b, y, q\rangle$, where the **address** $b \in B$, $y \in Y$, and the **successor sequence** q is a finite label sequence. Additionally, if $\langle b, y, q\rangle$ is a macro call, then so is $\langle b', y, q'\rangle$, where b' is any label and q' is any label sequence of the same length as q.*

The structure of a macro call core is left unspecified. Sequence q identifies the macro calls to which the run-time control may be transferred; any macro call with q longer than 1 is a conditional branch. Thus control flow, unlike data flow, is explicit in the model.

Definition 5.2 *Concerning any given macro call $c = \langle b, y, q\rangle$, we use the following notations:*
(a) *$\beta(c)$ denotes b.*
(b) *$\mu(c)$ denotes $\mu_Y(y)$.*

Thus, $\beta(c)$ is the address of c, and $\mu(c)$ is the name of the macro called by c.

Definition 5.3 *The set of finite non-empty macro call sequences is denoted by C^+.*

Note that we are interested only in non-empty macro call sequences. Next, you will see that an expansion result must indeed contain at least one macro call—which may in practice, however, be an atomic 'no-op' operation.

Definition 5.4 *We define what a **tree** is like. Moreover, for each tree t, we define macro call $\rho(t)$, which is the **root**; a label set $N(t)$, which is the set of the addresses of the **nodes**; and a label set $L(t)$, which is the set of the addresses of the **leaves**. Now, the set of trees is the smallest set Θ that meets both the following conditions:*
(a) *Each macro call c belongs to Θ. In this case, $\rho(c) = c$ and $N(c) = L(c) = \{\beta(c)\}$.*
(b) *Each pair $\langle c, z\rangle$ belongs to Θ if c is a macro call and z is a finite non-empty sequence whose elements $t_1, \ldots, t_m$ all belong to Θ, such that for each i in range $1, \ldots, m$ it holds that $\beta(c) \notin N(t_i)$, and for each two distinct i' and i'' in range $1, \ldots, m$ it holds that $N(t_{i'}) \cap N(t_{i''}) = \emptyset$. In this case, $\rho\langle c, z\rangle = c$, $N\langle c, z\rangle = \{\beta(c)\} \cup \bigcup_{i=1}^{m} N(t_i)$, and $L\langle c, z\rangle = \bigcup_{i=1}^{m} L(t_i)$.*

In Fig. 19 we have an example of a tree of the form $\langle c_0, \langle\langle c_1, \langle c_4, c_5\rangle\rangle, c_2, c_3\rangle\rangle$. The address and the follower sequence of each macro call are visible: macro calls (whose addresses are) *B1* and *B4* are conditional branches and *B5* is an unconditional branch. By Def. 5.4, all the nodes of a tree have unique addresses.

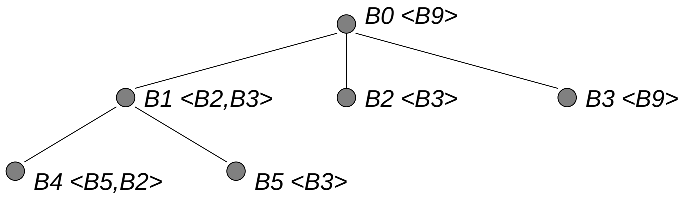

Figure 19: A tree.

Definition 5.5 *We define some relations between tree nodes. Supposing that c and c' are among the nodes of a given tree t, we have:*
(a) *c' is a **child** of c in t if t contains a subtree $\langle c, \langle t_1, \ldots, t_k, \ldots, t_m\rangle\rangle$ such that $c' = \rho(t_k)$.*
(b) *c' is a **parent** of c in t if c is a child of c' in t.*

(c) *c and c' are **siblings** in t if $c \neq c'$ and if there exists such a node of t that is a parent of both c and c' in t.*
(d) *c' is a **descendant** of c in t if the pair $\langle c', c\rangle$ belongs to the transitive closure of the child relation defined over the nodes of t.*
(e) *c' is an **ancestor** of c in t if c is a descendant of c' in t.*

The above terminology is most intuitive. It is easy to see that every non-root node has exactly one parent and is a descendant of the root, and that every non-leaf node has a non-empty set of descendants. (Of course, the notion of a subtree—employed in the definition of a child—includes even a "non-proper" one, i.e. the whole tree.)

Specification 5.6 *There is a set U of **user settings**.*

The generation of the expansion tree cannot be independent of the code that initially surrounds the root macro call, but the user should be able to ignore the details and to take into account only the essential properties of the surroundings. Hence, it is required that the contents of the metastatic interface of the root macro call are explicitly specified by the user.

Definition 5.7 *An **expansion tree** or, more shortly, an **x-tree** is a pair $\langle u, t\rangle$, where $u \in U$ and t is a tree. Furthermore, for each x-tree $x = \langle u, t\rangle$, we simply define $\rho(x) = \rho(t)$, $N(x) = N(t)$, and $L(x) = L(t)$; in a similar fashion, even the "family relations" of Def. 5.5 are extended to concern x-trees.*

The structure of the expansion tree to be generated depends strongly on the user setting (actually, below we will argue that the structure depends only on the user setting and the root macro call).

5.2 Macro call environment

The conditional expansion of the macro call takes place against its environment. Here we define the extent of the environment.

Specification 5.8
(a) *There is a set E of **environments**.*
(b) *There is an **initial extractor** function $\xi^0 : U \times C \to E$.*
(c) *There is a **differential extractor** function $\xi : E \times C^+ \times C \to E$.*

Thus, 'extraction' means the identification of the macro call environment. (The environment implicitly fixes the contents of the metastatic macro call interface. In this section we ignore the explicit building of the metastatic interface performed by the macro expander.)

Definition 5.9 *Each node c of an x-tree $x = \langle u, t\rangle$ has an unique environment $\epsilon(x, c)$.*

(a) *If c is the root of x, then $\epsilon(x, c) = \xi^0(u, c)$.*
(b) *Otherwise, x must contain such a subtree $\langle c', \langle t_1, \ldots, t_m \rangle\rangle$ that sequence $\langle \rho(t_1), \ldots, \rho(t_m)\rangle$ contains c; then, $\epsilon(x, c) = \xi(\epsilon(x, c'), \langle \rho(t_1), \ldots, \rho(t_m)\rangle, c)$.*

The above definition is recursive: part (b) determines the environment of a node in an x-tree by referring to the environment of the parent of that node, whereas part (a) is the recursion base.

Proposition 5.10 *Suppose that an x-tree x' is constructed by replacing in an x-tree x a single macro call c with another call c'. Furthermore, suppose that x (and therefore even x') also contains a macro call c'' different from c. Now, if the environment of c'' is in x' different from the one in x, then in x exactly one of the following holds:*
(a) *c'' is a descendant of c.*
(b) *c'' is a sibling of c.*
(c) *c'' is a descendant of a sibling of c.*

Proof sketch: This is a straightforward consequence of Def. 5.9.

Note that Proposition 5.10 is actually illustrated by Fig. 7 on page 25 and commented in Secs. 3.4.2 and 3.4.3. Consider also what would happen if the reading of part (b) of Def. 5.9 were changed (of course, the domain of the differential extractor ξ would also have to be appropriately changed). If the definition $\epsilon(x, c) = \xi(\epsilon(x, c'), \langle \rho(t_1), \ldots, \rho(t_m)\rangle, c)$ were replaced with $\epsilon(x, c) = \xi(\epsilon(x, c'), c)$, then we would lose context-sensitivity and thus propagativity. If the replacement were $\epsilon(x, c) = \xi(\epsilon(x, c'), \langle t_1, \ldots, t_m \rangle, c)$, instead, then we would lose order-independence (which will be established by Proposition 5.23) and thus stability.

5.3 Notion of harmoniousness

The expansion result of a macro call should respect the environment of the macro call. This requirement is reflected by the notion of a harmonious expansion tree—in addition, we will use this notion for establishing a technically convenient, that is, deterministic, labeling scheme for the macro calls within a single expansion tree.

Specification 5.11 *There is a bijective* ***demultiplexer*** *function $\delta : B \times \mathbf{Z} \to B$, where $\mathbf{Z}$ is the set of integers.*

The existence of the demultiplexer function δ is guaranteed because the set B of labels is countably infinite: since sets $B \times \mathbf{Z}$ and B thus have the same cardinality, bijections do exist.

Definition 5.12
(a) *A label b is* ***pristine*** *if there is a label b' and an integer $i \leq 0$ such that $\delta(b', i) = b$.*
(b) *The* ***progeny*** *of a label b is the smallest set of labels that meets the following condition: each given label b' belongs to the set if there is a label b'' and an integer $i > 0$ such that $\delta(b'', i) = b'$, and either $b'' = b$ or b'' belongs to the set.*

By requiring the labels occurring in the root to be pristine, they are prevented from reappearing among the node addresses, as it will be shown below: the addresses of the nodes in a harmonious tree will be required to belong to the progeny of the root address.

Specification 5.13 *There is a* ***harmony predicate*** *function $\chi : E \times C^+ \to \{0, 1\}$. More specifically, χ has the properties to be given in Spec. 5.15.*

The further properties of the harmony predicate will be specified by restricting the cases to which it applies. Before that refinement, we give an overall characterization of the usage of the predicate.

Definition 5.14
(a) *A macro call sequence $s \in C^+$ is* ***harmonious*** *with an environment e if $\chi(e, s)$ holds.*
(b) *An x-tree x is* ***harmonious*** *if the address and the successor sequence of its root include only pristine labels, and if for each subtree $\langle c, \langle t_1, \ldots, t_m \rangle \rangle$ occurring in x, the macro call sequence $\langle \rho(t_1), \ldots, \rho(t_m) \rangle$ is harmonious with environment $\epsilon(x, c)$.*

To some extent, the expander is able to verify that the selected macro version respects the macro call environment. In other words, the expander accepts only such a version that it can transform into a harmonious macro call sequence.

Specification 5.15 *The harmony predicate obeys the following three principles:*
(a) *The* ***labeling principle*** *states that if an x-tree x contains a macro call c, and if macro call sequence $\langle c_1, \ldots, c_m \rangle$ is harmonious with $\epsilon(x, c)$, then for each i and j in range $1, \ldots, m$ it is the case that $\beta(c_i)$ belongs to the progeny of $\beta(c)$ but not to the progeny of $\beta(c_j)$.*
(b) *The* ***copying principle*** *states that if a harmonious x-tree x contains a macro call $c = \langle b, y, q \rangle$, then for any integer $i > 0$ the singleton sequence $\langle \langle \delta(b, i), y, q \rangle \rangle$ is harmonious with $\epsilon(x, c)$.*
(c) *Let a macro call sequence $s = \langle c_1, \ldots, c_k, \ldots, c_m \rangle$ be harmonious with an environment e. Let there also be a macro call sequence $\langle c'_1, \ldots, c'_n \rangle$ that is harmonious with $\xi(e, s, c_k)$. Then for any macro call d, let $\tilde{d}$ denote the macro call that results when in d each occurrence of $\beta(c_k)$ is replaced with $\beta(c'_1)$. Now*

*the **merging principle** states that even* $\langle \tilde{c}_1, \ldots, \tilde{c}_{k-1}, \tilde{c}'_1, \ldots, \tilde{c}'_n, \tilde{c}_{k+1}, \ldots, \tilde{c}_m \rangle$ *is harmonious with* e.

The labeling principle effectively guarantees that the addresses in the macro expansion result will remain unique. The copying principle tells that, informally, no macro call violates its own environment. The merging principle implies that no lower-level macro call can explicitly violate the environment of the root, no matter how long the expansion continues. Note especially that the copying and merging principle do not contradict the labeling principle.

5.4 Linking and order-independence

By 'linking' we mean the basic expansion action which provides a macro call instance with an appropriate implementation version of the called macro. Thus, the expansion tree grows larger through linking operations.

Specification 5.16 *There is a set* P *of **test predicate** functions* $p : E \to \{0, 1\}$.

Using test predicates, the macro writer can examine the environment of a macro call, in order to find the best possible macro version for the particular environment.

Specification 5.17 *There is a set* V *of **macro versions**.*

Each macro definition consists of a sequence of alternative implementation versions.

Definition 5.18
(a) *A **macro definition** is any pair* $\langle m, w \rangle$*, where* $m \in M$ *and* w *is a finite sequence of pairs of the form* $\langle p, v \rangle$*, where* $p \in P$ *and* $v \in V$.
(b) *A finite set* Λ *of macro definitions is a **library** if for any two distinct* $\langle m', w' \rangle$ *and* $\langle m'', w'' \rangle$ *in* Λ*, it holds that* $m' \neq m''$.

Each macro version is thus guarded by a condition. The versions are listed in the order of decreasing priority (see Def. 5.20); this priority scheme makes the expansion result deterministic. All the macros for which there is no definition in the given library are interpreted as atomic placeholders.

Specification 5.19 *There is a **transformer** function* $\tau : E \times V \to C^+$.

The selected macro version must be adapted to the particular environment; this transformation is performed by the macro expander. (In the case of machine-level computer program synthesis, we suggest that this transformation includes the intraclass storage allocation.)

Definition 5.20 *Let an x-tree x contain a node c, and let e denote $\epsilon(x, c)$. Then a finite non-empty macro call sequence $s = \langle c_1, \ldots, c_n \rangle$* ***realizes*** *macro definition $\langle m, w \rangle$ at c in x if w contains such an element $\langle p, v \rangle$ that all the following conditions are met:*
(a) *$p(e)$ holds.*
(b) *$s = \tau(e, v)$.*
(c) *s is harmonious with e.*
(d) *For each integer i in range $1, \ldots, n$ it holds that $\delta(\beta(c), i) = \beta(c_i)$.*
(e) *Among the elements of w that precede $\langle p, v \rangle$, there is no such $\langle p', v' \rangle$ that both $p'(e)$ holds and $\tau(e, v')$ is harmonious with e.*

A macro version thus results in a macro call sequence that realizes the macro definition only if, first, the version-specific test is satisfied by the environment and, second, the expander is able to transform the version into a harmonious macro call sequence. Note especially that condition (d) of Def. 5.20, which uniquely fixes the addresses of the macro calls in the resulting sequence, does not contradict the labeling principle stated in Spec. 5.15; precisely that principle guarantees that the addresses can be determined locally in the way suggested here. (For a more refined characterization of transformation failures in the case of machine-level computer program synthesis, you may want to consult Sec. 3.3.3.)

Definition 5.21 *An x-tree x' is an* ***immediate derivative*** *of an x-tree x with respect to some library Λ, denoted as $x \Longrightarrow_\Lambda x'$, if there is a leaf c of x, $\langle m, w \rangle \in \Lambda$, and $s \in C^+$ such that all the following conditions are met:*
(a) *x is harmonious.*
(b) *$\mu(c) = m$.*
(c) *s realizes $\langle m, w \rangle$ at c in x.*
(d) *x' is obtained from x by replacing c with $\langle c, s \rangle$.*

Each single expansion step produces an immediate derivative of the previous expansion tree. The labeling principle guarantees that the addresses of the nodes in the x-tree remain unique.

Definition 5.22 *For each non-negative integer i, an x-tree x' may be a* ***derivative of degree*** *i of an x-tree x with respect to some library Λ, denoted as $x \Longrightarrow_\Lambda^i x'$. The actual definition is inductive:*
(a) *If x is harmonious, then $x \Longrightarrow_\Lambda^0 x$.*
(b) *If there is an x-tree x'' such that both $x \Longrightarrow_\Lambda^k x''$ and $x'' \Longrightarrow_\Lambda x'$, then $x \Longrightarrow_\Lambda^{k+1} x'$.*

Informally, expandability is the reflexive-transitive closure of one-step expandability.

Proposition 5.23 *Suppose that $x \Longrightarrow_\Lambda^i x'$ and $x \Longrightarrow_\Lambda^j x''$. Then there exist* *such x^*, k, and l that both $x' \Longrightarrow_\Lambda^k x^*$ and $x'' \Longrightarrow_\Lambda^l x^*$.*

This Church-Rosser-type property [12, 51, 87] guarantees that no expansion result sought after can be missed by a wrong selection of the leaf to be expanded next: the final expansion result (if it exists) is completely determined by the user setting and the root macro call.

Proof sketch: By Proposition 5.10, every such node whose structure may affect the environment of a given node is present in the expansion tree whenever the given node is: forthcoming expansion steps cannot ever change the node environment. Specifically, in part (b) of Def. 5.9, we have $\epsilon(x, c) = \xi(\epsilon(x, c'), \langle\rho(t_1), \ldots, \rho(t_m)\rangle, c)$, which means that the environment of a macro call c depends on no other parts of the possibly already existing subtrees t_i than their roots, which do exist whenever c exists. (Actually, the set of nodes that may affect the environment is precisely the set of nodes whose presence is guaranteed!) For example, in Fig. 20 only the white-colored nodes contribute to the environment of node *N1*: this environment is determined by nodes *N1*, *N2*, and *N3*, and the environment of node *N0*. As the structure of the subtree originating from a given node is fully determined by the macro name and the environment of the node, the structures of neighboring subtrees are independent of the order in which they are generated.

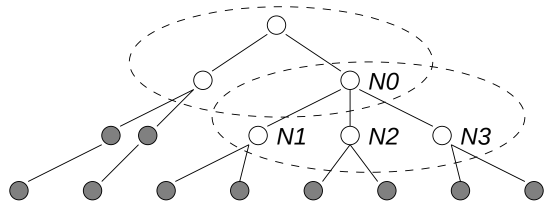

Figure 20: Extracting the environment of node *N1*.

Proposition 5.24 *Suppose that both $x \Longrightarrow_\Lambda^i x'$ and $x \Longrightarrow_\Lambda^j x'$. Then it is the case that $i = j$.*

This proposition states the full insignificance of the expansion order: no wrong choice of a leaf to be expanded next can even delay our reaching the final expansion result (whenever such one exists).

Proof sketch: Each expansion step increases the number of nodes in the x-tree. Moreover, the precise increase depends only on the environment at the particular leaf, which is independent of the expansion order. This means that whenever two expansion step sequences result in the same x-tree, they must comprise the same node number increases (even if possibly differently ordered) and thus be of the same length.

5.5 Merging and global optimizations

Whereas linking grows the expansion tree, merging contracts it into a more polished form by removing the intermediate nodes different from both the root and the leaves. The order-independence of the linking operations presupposes that they are performed without intervening merging operations. On the other hand, in practice the integration of an add-on global optimization routine has to be facilitated by a preliminary full merge of the tree.

Definition 5.25 *Let X^1 be the set of* ***wide*** *x-trees, that is, the set of harmonious x-trees that have a single non-leaf node (which thus must be the root).*

The notion of wideness reflects the fact that typical program representations do not intermix multiple abstraction levels as freely as our x-trees.

Definition 5.26 *The label which is the* ***image*** *of a given label b with respect to a given x-tree x is defined recursively as follows:*
(a) *If b does not belong to $N(x)$, or if b belongs to $L(x)$, then the image of b is b itself.*
(b) *Otherwise, x contains a subtree of the form $\langle c, \langle t_1, \ldots, t_m \rangle\rangle$ such that $b = \beta(c)$. Now the image of b is $\beta(\rho(t_1))$.*

The image function thus maps the address of each node to the address of the leaf determined by primogeniture.

Definition 5.27 *The* ***flattening*** *of an x-tree $x = \langle u, t\rangle$ is the x-tree $\bar{x}$ defined as follows:*
(a) *If t is a macro call $c = \langle b, y, q\rangle$, then $\bar{x} = \langle u, \langle\langle b, y, q\rangle, \langle\langle \delta(b, 1), y, q\rangle\rangle\rangle\rangle$.*
(b) *Otherwise, $\bar{x} = \langle u, \langle \rho(t), s\rangle\rangle$, where s is the macro call sequence that is obtained by first concatenating all the leaves of t, in the same order as they appear in t, and then replacing the occurrences of each label with its image (with respect to x).*

In case (b), flattening removes the intermediate nodes from the tree and redirects the remaining references to them to point to their image leafs. (The leaf concatenation order is indeed obvious; see Def. 5.4.)

Proposition 5.28 *The flattening of a harmonious x-tree is a wide x-tree.*

Proof sketch: The most challenging part is the establishment of harmoniousness. If the original tree is a macro call, then the copying principle applies. Otherwise, the full flattening can be carried out by repeated elementary flattening operations, each of which lifts a single macro call sequence upwards in the tree. By the merging principle, each such lifting preserves harmoniousness.

Definition 5.29 *For any integer k, an x-tree x' is a k-**finalization** of an x-tree x with respect to some library Λ if there exists an x-tree x'' such that all the following conditions are met:*
(a) $x \Longrightarrow_{\Lambda}^{k} x''$.
(b) *For each leaf c of x'' and for each macro definition $\langle m, w\rangle$ in Λ, it holds that $\mu(c) \neq m$.*
(c) *x' is the flattening of x''.*

A k-finalization is the flattening of a full-blown expansion result, and its existence implies that the full-blown x-tree is achieved by precisely k expansion steps.

Proposition 5.30 *If there exists a k-finalization of a given x-tree with respect to some library, then this k-finalization is wide and unique, and there does not exist a l-finalization (of the same x-tree and with respect to the same library) for any $l \neq k$.*

Proof sketch: This is a direct consequence of Proposition 5.23, Proposition 5.24, and Proposition 5.28.

Specification 5.31 *There exists a set G of **global optimization** functions $g : X^1 \to X^1$. Specifically, for each $g \in G$ and $\langle u, \langle c, s\rangle\rangle \in X^1$, there is a macro call sequence s' such that $g\langle u, \langle c, s\rangle\rangle = \langle u, \langle c, s'\rangle\rangle$.*

Note that even doing nothing would be valid as a global optimization. (In the case of machine-level computer program synthesis, global optimizations would be such procedures as, say, global common subexpression analysis.)

Definition 5.32 *A **generator** is any pair $\langle \Omega, \Gamma\rangle$, where Ω is a finite set of libraries and Γ is a finite set of global optimizations.*

Code generation is constituted by macro expansion augmented with intermediate global optimizations; see Fig. 21.

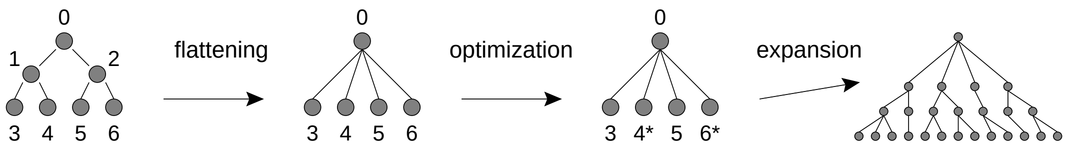

Figure 21: Integrating an intermediate global optimization.

Definition 5.33 *The set of x-trees that are **generated** from an x-tree x by a generator $\langle \Omega, \Gamma\rangle$ is the smallest set Ξ that meets the following conditions:*
(a) *If $x' = x$ or if x' belongs to Ξ, and if there is such k that x'' is the k-finalization of x' with respect to some library in Ω, then x'' belongs to Ξ.*

(b) *If* x' *belongs to* Ξ*, and if* $x'' = g(x')$ *for some* $g \in \Gamma$*, then* x'' *belongs to* Ξ.

Proposition 5.34 *If an x-tree* x' *is generated from an x-tree* x *by a generator, then* x' *is wide.*

Proof sketch: The proposition follows trivially from Def. 5.33.

6 Related work

The basic conceptual structure of a compiler [2, 34], shown in Fig. 22, consists of *analysis* and *synthesis* phases: the *front end* first parses the source program written by the user into a suitable *intermediate representation* (IR), from which the *back end* then generates the machine-level program to be run on the target processor. Note that here we deliberately use the terms 'parsing' and 'code generation' in a rather wide sense: for example, our parsing task covers at least lexical, syntactical, and semantical analysis (which are typical subphases of the analysis phase).

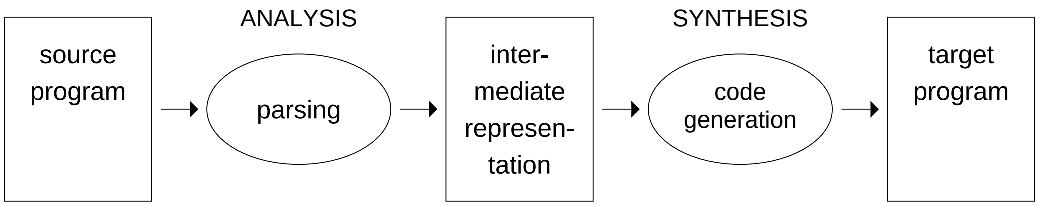

Figure 22: Basic compiler structure.

There are two major reasons motivating the definition and adoption of an intermediate representation [2, p. 463]. First, retargeting the compiler should become easier: one may hope that only the back end needs to be rebuilt. Second, there is the possibility to optionally perform optimizing transformations [10] within the IR, as shown in Fig. 23: such optimizations should not require modifications in the front or back end.

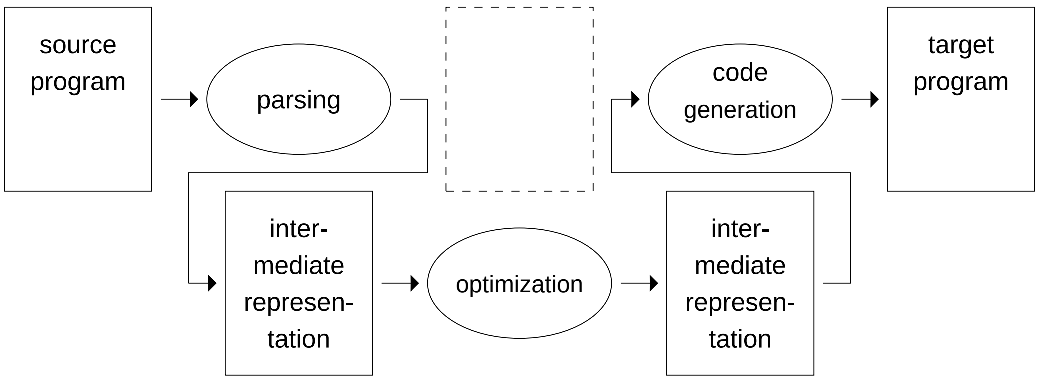

Figure 23: An optimizing compiler.

Sometimes it may be profitable to introduce even more than one IR: in Fig. 24 we have two IRs. We define code generation as a transformation that converts the program from one representation into another and involves machine-dependent commitments. In practice, there are thus at least as many code generation phases as there are IRs.

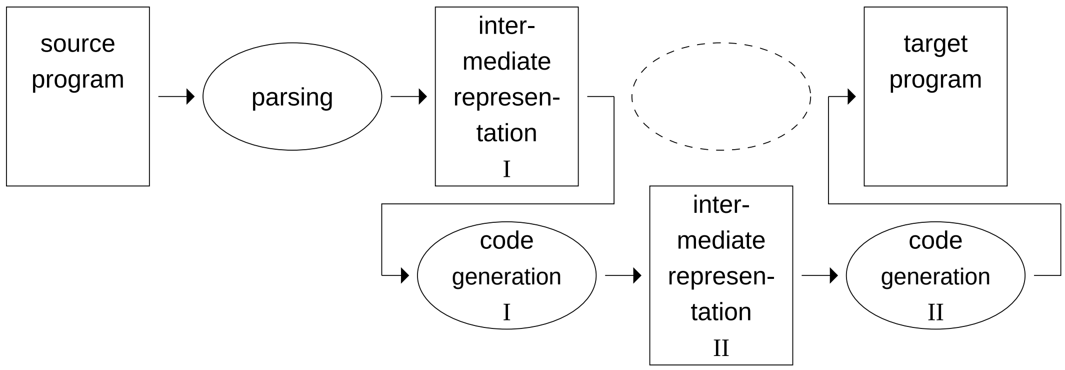

Figure 24: Two-phase code generation.

The outline of the rest of this section is as follows. We are interested in such approaches in compiler design whose primary concerns include run-time efficiency; below we will examine (machine-oriented) source languages in Sec. 6.1, intermediate representations in Sec. 6.2, and techniques for implementing code generation in Sec. 6.3.

6.1 Source languages

The first programming languages different from machine languages were assembly languages. Macros were an early addition to the assembler facility [45, 75, 56, 88]. From our present viewpoint, particularly interesting are the *Ferguson-type meta-assemblers* [33, 90, 88]: a meta-assembler is a retargetable tool for assembler construction, and a Ferguson-type one is retargeted by writing a macro definition for each machine instruction of the new processor. Thus, our ReFlEx system might be seen as a Ferguson-type meta-assembler that has been augmented with the strong support for modular hierarchy.

A major obstacle to the approval of the first high-level language compilers was the loss of transparency. Potential compiler users could not estimate how much the run-time efficiency would degrade due to the compiler use, and the only way to assure them was by getting them to carry out experiments and by making sure that no alarming results would ever be produced. Accordingly, when the first Fortran system became available in 1957 [8], the main concern of the implementors was the requirement of consistently high output code quality [9], and the compiler consequently featured surprisingly powerful optimizations—even global program flow analysis was included [60]. But the extreme significance of optimization soon decreased, as noted by J. W. Backus and W. P. Heisig in 1968 [9]. Nowadays, new high-level languages are often designed to be, first of all, user-friendly, rather than ultimately efficient in terms of execution time and memory space.

The principles of structured programming [27, 22] proposed in the 1960's were soon applied even to assembly languages. Block structure and high-level control constructs (such as *if-then-else* and *while-do*) were introduced, but the data structure and operation one-to-one correspondence were still retained: the programmer was able to select the precise hardware registers and machine instructions. A pioneering example of such a *structured assembly language* was the PL360 language [101] for the IBM 360 computers, which N. Wirth designed to give the look-and-feel of a high-level language without sacrificing the efficiency of a machine-level language [102, p. 11]. The same idea was ported to other machines: for example, a similar language for the Zilog Z80 microprocessor is described in [13]. It has been demonstrated [97, 86] that at least some structured assembly languages can be implemented with macro assemblers.

Around 1970, there was considerable interest in designing *extensible* high-level languages [18, 91]; the extensions were supposed to be defined and implemented by the end users, often through macros (possibly in a machine-dependent fashion [5]). This activity was accompanied by similar developments in the field of fully machine-specific languages [31]: for an elaborate example, B. N. Dickman [26] presents a macro-based system that is capable of automatic "intraclass" register allocation within a basic block according to explicit live variable information provided by the user. Additionally, many researchers [16, 17, 46] proposed the use of general-purpose macro expanders as an implementation mechanism for special-purpose high-level languages (tailored to some particular application area): the suggested advantage of this approach was the easy portability of the implementation—whose run-time efficiency, however, was likely to remain rather poor.

Still in the 1970's, many high-level programming languages were designed [96, 21] specifically to be used in *system programming*, e.g. in the construction of compilers and operating systems. In such tasks it is certainly worthwhile to strive for efficiency (note that practically all embedded computer programs can be seen as system programs). In contrast to structured assembly languages, system programming languages are (at least relatively) machine-independent, which should promote the portability of the system programs. Two examples of system programming languages are Bliss [105, 104] and C [57, 58]. Although the Bliss definition is machine-independent, it is intentionally strongly geared towards the DEC PDP-10 architecture and cannot be implemented as efficiently on other machines. The C language definition, in contrast, is not geared towards any particular machine. Nevertheless, C programs are not strictly machine-independent: as C ignores some machine-specific details such as overflow handling conventions, running a single program on two different machines may produce different results. But this very laxity is perhaps the decisive factor behind the success of C (and the UNIX operation system written in C): there is only a small set of language elements, and these elements are likely to match well enough with the basic features offered by most target architectures. (In essence, C demonstrates the power of the *downward abstraction* principle; see [47, 26, 89], for instance.)

6.2 Intermediate program representations

The compiler writer has to determine both the structure and the semantics of the intermediate representation(s). Structural possibilities are presented in [2, 34] and include linear lists, trees, directed acyclic graphs, and unrestricted graph formats; the semantics problem seems to be the more interesting one. An important question is whether the IR should have a fixed semantics or whether it should be a *language schema* [3, 48] without built-in semantics (for a theoretical discussion on language schemas, see [73], for example). Clearly, our ReFlEx macro language can be seen as a language schema, as it has no built-in primitive operations.

As the primary motivation for the introduction of an IR is to improve retargetability, the IR semantics should ideally be independent of both the source and target language (i.e. the target processor). Accordingly, already in the 1950's many researchers tried to define a single universal IR language, traditionally referred to as UNCOL (UNiversal Computer Oriented Language) [94, 93, 92]. But similarly to the analogous *interlingua* approach in the machine translation of natural languages [53, 6], the theoretically most interesting UNCOL approach has been slow to find significant practical applications.

Some later, less ambitious approaches have proved to be much more successful:

- Using an IR that is tailored to the source language but fully independent of the target language [77, 84].
- Using an IR that is an *intersection language* with only a relatively small set operations [23]. This small set should be chosen so that it is nevertheless, first, likely to be supported by a relatively broad class of contemporary target architectures, and, second, likely to support a relatively broad class of contemporary source languages.
- Using an optimization-oriented machine-specific IR (which is fully independent of the source language) [14], perhaps in addition to a higher-level IR serving to isolate the machine-independent front end from the back end.

6.3 Code generation techniques

Overall information about state-of-the-art code generation methods can be found in the standard compiler textbooks [2, 34], which are from the late 1980's. Additionally, retargetable code generation tools were surveyed in the early 1980's by Ganapathi, Fisher, and Hennessy [40], and by Lunell [72]. In particular, Lunell [72, Ch. III.2] separates the *automation* and *support* ideals set for different code generator writing systems: many approaches aim at maximum automation, whereas some strive for a system that only supports the human code generator writer. Below, partially following the classification used in [40], we examine *interpretative* methods, methods based on a *context-free grammar*, and heuristic *pattern matching* methods.

The interpretative tools can be associated with the support ideal. The earliest interpretative code generation method was the use of a general-purpose macro expander [16, 17, 20, 15, 76]. Two early special-purpose programming languages tailored to code generation (which were never developed into a fully retargetable form) were described by Elson and Rake [28] and by Wilcox [98]; later ones include Fraser's language [37] and our proposal. The languages of [28, 98, 37] are procedural, whereas ours is functional: there are no side effects. The intermediate program representations examined and manipulated in [28, 37] consist of trees; in [98] a linear operation sequence is used; we establish an abstract interface for hiding the program representation and the built-in algorithms processing it. When the intermediate program representation is a linear sequence, it is straightforward to keep track of the machine register contents within a basic block (i.e. a straight-line code segment), in the fashion addressed above in conjunction with macro-based code generators. Our goal is more ambitious: we want to be able to utilize all the information about the run-time context that is available already at compile time. In particular, we want to be able to look ahead as well as back, and beyond basic-block boundaries.

The Graham-Glanville method [42, 43, 44, 50] uses context-free grammars and LR parsing for code generation (the obvious extension direction is to use attribute grammars, as suggested in [39]). Being fully transparent, like the corresponding syntax analysis performed by the compiler front end, the method is not inconsistent with the support ideal but it differs from the earlier interpretative approaches by being highly declarative. On the other hand, the intermediate representation accepted by a Graham-Glanville code generator must be a rather low-level one: many of synthesis decisions must be made already prior to the code generation. Though our approach shares this same restriction, it strives for a wider scope by its intuitive context-sensitivity.

Many modern code-generator generators [1, 83, 30, 11, 38] use tree pattern matching and dynamic programming: the compiler writer specifies a collection of code generation patterns with associated costs, and the code generator tries to find a set of patterns that covers a given tree in the intermediate program representation with a minimum total cost. The operation is thus heuristics-driven, sharing the automation ideal. This method is neither fully transparent nor particularly stable: when a single pattern or its associated cost is about to be modified, it may be difficult to predict the overall consequences.

7 Conclusion

Traditionally, macro expansion systems employ global variables to achieve some context-sensitivity. But the use of global variables has some significant drawbacks. First, the Church-Rosser property is lost: the macro expansion result becomes dependent on the expansion order. Second, the gained context-sensitivity is very limited: information about the context can typically be passed only from left to right, in accordance with the customary left-to-right and depth-first expansion order.

Our main contribution is the formal model of Sec. 5, which describes a modular and hierarchical abstraction mechanism that we claim to be reasonably universal and transparent, as defined in Sec. 2.3. In particular, the model establishes context-sensitivity without violating order-independence, i.e. the Church-Rosser property; our idea of context in the case of machine-level computer program synthesis is characterized in Sec. 3. Order-independence and context-sensitivity effectively offer stability and propagativity, respectively, as defined in Sec. 2.3. But most importantly, the order-independence means that the macro expansion language is functional rather than procedural: for instance, extensively parallel implementations become possible.

The prototype implementation dealt with in Sec. 4 is admittedly extremely simple and provides no evidence of practical usefulness. Indeed, our next goal will be an extended prototype that supports most of the features mentioned in Sec. 3.2. After these fully experimental systems, we should be able to try to construct a code generation tool for a real processor. Most promising contemporary target processors seem to include the "purest" DSPs, i.e. the stripped-down fixed-point number crunchers with limited functionality, low cost, and high speed—such as AT&T DSP1610 [7]. Still, the retargetability of the tool should be preserved, and furthermore, demonstrated by providing the class of supported target architectures with an explicit characterization (in principle, such a characterization might even serve as a constraint on future processor design).

A ReFlEx 1.0 macro language

This appendix systematically discusses the features of the ReFlEx 1.0 macro language. First, Secs. A.1 and A.2 introduce some general concepts. Next, Secs. A.3–A.5 deal with the rule file, which contains the macro definitions. Finally, Sec. A.6 covers the expansion source, i.e. the code generation input. Many secondary features of the full ReFlEx 1.0 language are still, for simplicity, ignored. (See [63] for a complete reference to the language.)

The rule file provided by the user must be a single physical file at the time when the macro expansion session is started. The information contained in the rule file cannot be updated during the session; in particular, new macro definitions cannot be created. Having processed the rule file, ReFlEx prompts the user to type in an expansion source. The expansion source should consist of a macro call and a specification of its metastatic interface. Given the expansion source, ReFlEx expands the macro call in it, and then prompts for another task.

A.1 Taxonomy of integer, cell, and label designators

The elementary objects processed at expansion time include *integers*, *storage cells*, and *code labels*. Such an expression written in the rule file or expansion source that may at expansion time produce an integer is called an *integer designator*; similarly, there are also *cell designators* and *label designators*. Being an integer, cell, or label designator is actually a property of each particular occurrence of the expression: if an expression occurs in two different contexts, it may be that only one of the occurrences is a valid integer designator. In other words, an expression occurrence is a valid integer (or cell or label) designator if and only if:

- its syntax is appropriate; and
- it appears in an appropriate context.

Furthermore, as ReFlEx classifies the designators orthostatically (that is, already before the macro expansion), a single expression occurrence may be, say, both an integer designator and a cell designator. Whether such an ambiguous designator metastatically (that is, at expansion time) really produces an object of the anticipated kind, cannot generally be orthostatically determined. Obviously, there are three possible cases of designator mismatch: if a valid integer designator actually turns out to produce a non-integer (i.e. a cell or a label), ReFlEx simply takes zero as the value produced by the designator; for the other two cases, see Sec. A.2.2.

Each designator is syntactically either a *literal*, a *reference*, or a *form call*.

A.1.1 Integer, cell, and label literals

The programmer uses literals in the expansion source, and references and form calls in the rule file; as a natural exception, integer literals may freely be used even in the rule file. The expansion result produced by ReFlEx contains literals only.

An *integer literal* is any digit sequence optionally preceded by a plus or minus sign. Here are some examples of valid integer literals:

```
7                 + 7                 - 7
```

A *cell literal*, then, consists of a storage class name and an integer literal serving as a zero-based index within the class (if the storage class contains only one cell, the index can be dropped off):

```
M[0]              M[1]                M[10]
```

Finally, any identifier can be used as a *label literal*. (See Sec. A.1.2 for the definition of an identifier.)

A.1.2 Integer, cell, and label references

As mentioned above, integer, cell, and label references can be used in the rule file but not in the expansion source. Any integer, cell, or label reference is constituted either by an identifier or a reserved word. An *identifier* is a maximal sequence of alphanumeric characters that is different from the reserved words and does not begin with a digit (alphanumeric characters include letters, digits, and the underscore '_'; the distinction between uppercase and lowercase letters is significant). Each *reserved word* consists of uppercase letters only. (Even if we do not list the reserved words here, they can be easily distinguished from the identifiers used in this report, because the latter either contain non-uppercase characters or consist of a single uppercase letter.)

First, let us look at form definitions. Inside a form definition, the form parameters, which are identifiers, are valid as integer references, as cell references, and even as label references.

In our first example, form parameter `a` is used as an integer reference on the right-hand side of the form definition:

```
Sqr(a) = _Mul(a,a);
```

In the next example, the form parameter is used as a cell reference in order to decide whether the referred cell belongs to one of certain storage classes:

```
Convenient(x) = Or(?R(x),?A(x));
```

In addition to form definitions, there are macro definitions, of course. Inside a macro definition,

- the input parameters are valid integer references;
- the data parameters and the data temporaries are valid cell references; and
- the exit parameters, the label temporaries, and reserved words `THIS` and `NEXT` are valid label references.

Here, input parameter `x` is used both as an integer reference and as a cell reference, whereas data temporary `t` may only be used as a data reference:

```
t_incr(x > y) {
  integer: TEST Int(x); { t_move(_Add(x,1) > y); }
  cell: USE R[t]; { t_add(x,1 > t); t_move(t > y); }
}
```

Of the label references in our final macro definition, `l` is an exit parameter and `s` is a label temporary (associated with an empty statement):

```
BRANCH t_branch(x) [l] {
  nop: TEST &(l, NEXT); { }
  impl: { BRANCH t_1(x) [s]; BRANCH t_2(x) [l]; s: ; }
}
```

The test involving `NEXT` above detects such anomalous cases in which the branch target is the code location immediately following the current macro call: `THIS` and `NEXT` mark the beginning and end of the expansion result of the current macro call, respectively.

A.1.3 Form calls

Each form call is a valid integer designator but invalid as a cell or label designator. More on forms can be found in Sec. A.2.

A.2 Forms

Form calls are integer designators. At expansion time, each form call is evaluated by a special interpreter (which is an integral part of ReFlEx) according to the definition of the particular form. There is a set of predefined *primitive forms*, which consists of nine *autonomous forms* and six *diagnostic forms*. Additionally, the user may define new *compound forms* in the `HEADER` part of the rule file.

Form *argument designators* are those expressions that are visible in the form calls of the rule file. A form argument designator may be an integer designator, a cell reference, or a label reference (thus, cell and label literals are ruled out); an exception is that a macro temporary (even though it is certainly either a cell reference or a label reference) can never be used as a form argument designator. Form *arguments*, then, are the integers, cells, and labels that replace the argument designators at expansion time. Thus, each form argument is the metastatic evaluation result of the corresponding orthostatic argument designator.

A.2.1 Autonomous primitive forms

The argument designators of autonomous primitive forms must be integer designators. If a valid form argument designator still, at expansion time, turns out not to represent an integer, then ReFlEx uses zero as the metastatic argument, as stated in Sec. A.1.

The autonomous primitive forms can be divided into three categories:

- `_Abort(x,y,z)`, which raises an exception.
- `_If(x,y,z)`, which implements lazy argument evaluation.
- *Arithmetic-logic* primitive forms (see below).

Instead of returning, each call of `_Abort` aborts the expansion immediately. No expansion result is produced, but the three arguments of the `_Abort` call are passed to the user as an explanation.

The `_If(x,y,z)` call is processed similarly to the `x?y:z` expression of the C language. That is, the following steps are taken:

1. The first argument is evaluated.
2. If the result is nonzero: the second argument is evaluated, and the resulting value is returned as the value of the call.
3. Otherwise: the third argument is evaluated, and the resulting value is returned as the value of the call.

Other autonomous primitive forms always evaluate all their arguments, but either the second or the third argument of each `_If` call always remains unevaluated. Lazy evaluation of this kind makes even recursive compound form definitions feasible.

There are, in all, seven arithmetic-logic primitive forms, which are shown in Table 1. The rightmost column of the table consists of C language expressions, which fix the semantics of the primitives.

`_Add(x,y)`	addition	`x + y`
`_BNand(x,y)`	bitwise NAND	`~(x & y)`
`_BShl(x,y)`	bitwise shift	`(y > 0) ? (x << y) : (x >> -y)`
`_Div(x,y)`	division	`x / y`
`_Lt(x,y)`	less-than	`(x < y) ? 1 : 0`
`_Mul(x,y)`	multiplication	`x * y`
`_Nand(x,y)`	logical NAND	`(x && y) ? 0 : 1`

Table 1: Arithmetic-logic primitive forms.

A.2.2 Diagnostic primitive forms

The diagnostic primitive forms return information concerning the metastatic macro call interface constructed by the macro expander. Their argument designators must be cell or label references different from macro temporaries. However, if any argument designator still turns out to represent an integer, then the whole form call is taken to return zero. The same happens if an expected cell turns out to be a label, or vice versa. (Thus, one could say that "type checking" is here somewhat stricter than with autonomous primitives.)

Most of the diagnostic primitives accept any number of arguments, and there is an implicit conjunction between the arguments. For instance, the `?M(x)` form call checks whether `x` represents a cell that belongs to storage class `M`, and `?M(x,y,z)` checks whether all three of `x`, `y`, and `z` belong to `M`.

There are, in all, six diagnostic primitive forms, most of which may be further parametrized with a *storage class specifier* (such as '`M`' of the `?M(x)` form call above). We explain the usage of the six diagnostic primitives with the following examples:

`?(x)` Does `x` represent a cell?

`%(x,y)` Do `x` and `y` represent cells that belong to the same class?

`=(x,y)` Do `x` and `y` represent a single common cell?

`!(x)` Does `x` represent a free cell?

`#()` How many free cells are there?

`&(x,y)` Do `x` and `y` represent equivalent labels?

It should be noted that even `!(x)` and `#()` always report the situation at the currently processed macro call (as do the other diagnostic primitive forms), independently of the point at which they are placed inside the definition of the called macro. (This is the reason why macro temporaries, which do not have self-evident denotations outside the macro definition, are invalid as form argument designators.)

Finally, we make a few remarks about some individual diagnostic primitive forms:

- `&(x,y)` expects labels for arguments, while the others expect cells.
- `#()` accepts no arguments, while the others accept any number of arguments.
- `#()` returns a non-negative integer, while the others return either 0 or 1.
- `&(x,y)` does not accept a storage class specifier, while the others do.

A.2.3 Compound forms

Two most simple compound form definitions read as follows:

```
Dozen() = 12;
Gross() = _Mul(Dozen(),Dozen());
```

More complicated definitions are of course possible. The following definition of subtraction captures precisely the details of the two's complement representation for negative integers:

```
Sub(x,y) = _Add(x,_Add(_BNand(y,y),1));
```

The argument designators of a call of a compound form may freely be integer designators, cell references, or label references—macro temporaries excluded. As the first stage of the evaluation of a call of a compound form, and therefore even before the evaluation of the arguments, the interpreter replaces the call with the definition of the form called. This strategy supports lazy evaluation, because the definition may consist of an `_If` call.

Compound forms may be recursive, again because of the special property of the `_If` form. Here is a recursive definition for the factorial function:

```
Fact(x) = _If(_Lt(x,2), 1, _Mul(x,Fact(Sub(x,1))));
```

A.3 Rule file

A ReFlEx rule file contains both the target architecture description and the macro definitions. The file consists of four main parts, as shown in Fig. 25. These parts, each of which typically extends over several lines, are discussed below one at a time.

```
HEADER {     ...     }
STORAGE {     ...     }
SYSTEM {     ...     }
UTILITY {     ...     }
```

Figure 25: General structure of a rule file.

A.3.1 Compound form definitions

The `HEADER` part of the rule file contains the compound form definitions (see Sec. A.2.3), whose mutual order is insignificant. Fig. 26 shows a simple example.

```
HEADER {
  Not(a) = _Nand(a,a);
  And(a,b) = _If(a, _If(b,1,0), 0);
  Or(a,b) = _If(a, 1, _If(b,1,0));
  Int(a) = And(Not(?(a)), Not(&(a)));
  Lt(a,b) = And(_Lt(a,b), And(Int(a),Int(b)));
  Gte(a,b) = And(Not(Lt(a,b)), And(Int(a),Int(b)));
  Eq(a,b) = And(Gte(a,b), Gte(b,a));
}
```

Figure 26: A sample `HEADER` part.

A.3.2 Data storage declarations

The `STORAGE` part contains the data storage class declarations. For each storage class, name and size (i.e. the number of cells) are given. Fig. 27 shows an example with three storage classes.

A.3.3 Declarations of atomic macros

The `SYSTEM` part contains the declarations of the atomic macros. Each atomic macro can be seen as a ReFlEx-compatible model of a target machine instruction. The declaration of an atomic macro has a great deal in common with

```
STORAGE {
  M[1024];
  R[4];
  A[1];
}
```

Figure 27: A sample STORAGE part.

the definition of a composite macro: an atomic macro declaration consists of a macro exterior only, whereas a composite macro definition contains also a macro interior (see Sec. A.4 for the structure of a macro exterior). Again, Fig. 28 shows a simple example of a SYSTEM part.

```
SYSTEM {
  set(c > r) { TEST And(?R(r), And(Gte(c,-1024),Lt(c,1024))); }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) { TEST And(Or(?A(s),?R(s)), Or(?A(d),?R(d))); }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}
```

Figure 28: A sample SYSTEM part.

A.3.4 Definitions of composite macros

The UTILITY part finally contains the definitions of the composite macros; the mutual order of the definitions is insignificant. Fig. 29 shows a simple example of an UTILITY part, which contains two macro definitions. We must here assume that macros t_move, t_add, t_1, and t_2 in the figure are now atomic macros, because otherwise they should also be provided with a definition in this very UTILITY part; thus Figs. 28 and 29 are actually incompatible.

The structure of individual composite macro definitions is examined in detail in Sec. A.4.

A.4 Macro definition

ReFlEx supports macro hierarchy: within a composite macro definition, any atomic or composite macro can be called. In particular, both direct and

```
UTILITY {
  t_incr(x > y) {
    integer: TEST Int(x); { t_move(_Add(x,1) > y); }
    cell: USE R[t]; { t_add(x,1 > t); t_move(t > y); }
  }
  BRANCH t_branch(x) [l] {
    nop: TEST &(l, NEXT); { }
    impl: { BRANCH t_1(x) [s]; BRANCH t_2(x) [l]; s: ; }
  }
}
```

Figure 29: A sample SYSTEM part.

indirect recursion is allowed.

ReFlEx also provides extensive support for conditional macro expansion. Each macro definition comprises a number of alternative implementation versions. At expansion time, ReFlEx traverses through the version list in the order specified by the macro writer until it finds a version that matches the metastatic interface of the current macro call. If no such version is found, the macro expansion fails.

In Fig. 29 in Sec. A.3.4 we showed some simple macro definition. The general structure of a macro definition is presented in Fig. 30. As indicated in the figure, some of the items are optional.

$deviation_{opt}$ *name params* {
  $test_{opt}$

  version
  version
    ...
}

Figure 30: Structure of a macro definition.

The meaning of the items in Fig. 30 is as follows:

deviation Is the macro a branch of any kind? See Sec. A.4.1.

name Name of the macro being defined.

params Macro parameters (which represent storage cells and code labels). See Sec. A.4.2.

test Macro-specific test. See Sec. A.4.3.

version Definition of a macro version. See Sec. A.5.

The macro definition can be divided into the following parts:

- The *macro exterior* contains the *deviation*, *name*, *params*, and *test* items. It constitutes a partial specification of the macro call interface.
    - The *macro head* is a subset of the macro exterior consisting of the *deviation*, *name*, and *params* items. It specifies the structure of the orthostatic macro call interface.
- The *macro interior* contains the *version* items. It constitutes the macro implementation.

A.4.1 Control transfer

Concerning the control transfer after execution, ReFlEx macros are, for code readability, divided into four types by three distinct *deviation qualifiers*. All these qualifiers can be found in the following macro calls:

```
t_trans1(x > y);
BRANCH t_trans2(x > y) [l1,l2];
JUMP t_trans3(x > y) [l3];
l1: SLEEP t_trans4(x > y);
```

Macros of the `BRANCH` and `JUMP` types are the ones that may—explicitly—transfer the control to a remote location (specified by an exit parameter); macros of the default and `BRANCH` types are the ones that may—implicitly—transfer the control to the location immediately following (the code resulting from) the macro call itself. Thus `BRANCH` and `JUMP` macros are conditional and unconditional branches, respectively, whereas `SLEEP` macros typically represent non-terminating loops. Supposing that the macro calls above constitute a code fragment, the `t_trans4` call would be unreachable if it were not provided with the `l1` label temporary.

A.4.2 Parameters and arguments

There are two ways to classify *macro variables*. First, they can be divided into *data variables* and *label variables*. Second, they can be divided into *parameters* and *temporaries* (for the temporaries, see Section A.5.2). Because ReFlEx 1.0 does not support global variables, macro parameters are the only means for passing run-time data across macro boundaries.

There are both *data parameters* and *exit parameters*. A data parameter may be an *input* one, an *output* one, or an *input-output* one; exit parameters represent code labels which serve as branch targets. The parameters are specified in the macro head. We present some examples:

```
t_param1(x,y,z > z,w) {     ...     }
BRANCH t_param2(i,j) [l,m,n] {     ...     }
t_param3( > s) {     ...     }
JUMP t_param4() [t] {     ...     }
```

Note that the output parameters are separated from the input ones by a '>' token, which divides the data parameter list into two sublists. Understandably, the parameters that occur in both these sublists are the input-output ones. (The '>' separator is omitted if there are neither output nor input-output parameters.)

When a macro is called at expansion time, *macro arguments* stand for the macro parameters. A metastatic argument may be an integer, a storage cell, or a code label. With macro arguments, we do not employ such a special notion as the one of an input-output parameter with macro parameters—an output argument may simply simultaneously be even an input argument.

When a macro is called inside the definition of another macro, the macro writer has to provide the call with orthostatic *argument designators* that represent the metastatic arguments. The argument designators must meet the following constraints:

- An input parameter must be matched by an integer designator or by a data variable (of the calling macro definition).
- An output or input-output parameter must be matched by a data variable.
- An exit parameter must be matched by a label variable.

Here we provide sample calls for the macros whose heads were shown above:

```
t_param1(_Add(2,3),a,b > b,a);
BRANCH t_param2(u,u) [k,k,k];
t_param3( > a);
JUMP t_param4() [k];
```

A.4.3 Macro-specific test

Each macro definition may have a macro-specific test. This test is evaluated at expansion time, and it determines whether a call of the macro is acceptable as a part of the expansion-time realization of an upper-level macro call. For example, the `t_amacro` shown in Fig. 31 cannot be called unless the output argument belongs to storage class `A` or storage class `R`, and unless there are at least two cells currently free in class `R`.

```
t_amacro(x > y) {
  TEST And(Or(?A(y),?R(y)), Gte(#R(),2));
  implem1: TEST ?A(y); { t_a1(x > y); }
  implem2: TEST ?R(y); { t_a2(x > y); }
}
```

Figure 31: A macro definition with a macro-specific test.

As indicated in Fig. 31, the macro-specific test is constituted by an integer designator. The test is interpreted as a success if the designator at expansion time produces a nonzero value.

A natural implicit supplement to the macro-specific test is the following requirement: if two elements in the data parameter list of the macro definition are identical, then even the two corresponding arguments must be identical. As with the rest of the macro-specific test, the fulfillment of this requirement is checked only at expansion time.

In addition to macro-specific tests, there exist also version-specific tests. Two such tests are included in Fig. 31; see Sec. A.5 for more information.

A.5 Version definition

Each macro version specifies a possible implementation for a call of the macro being defined. The versions are listed in the order of decreasing priority. The criteria according to which ReFlEx either accepts or rejects an individual version are given in Sec. A.5.1 below.

The overall structure of each version definition is shown in Fig. 32. The meaning of the items in the figure is as follows:

name Name of the version being defined. (This name is more like a comment. It will be used nowhere else.)

test Version-specific test. See Sec. A.5.1.

temp Declaration of version-specific temporaries. See Sec. A.5.2.

stmt A statement. See below.

A *statement* consists of a unique label temporary, a macro call, and a ter-

$name$: $test_{opt}$ $temp_{opt}$ {
  $statement$
  $statement$
    ...
}

Figure 32: Structure of a version definition.

minating semicolon. Both the temporary and the macro call are optional (a statement is *empty* if it does not contain a macro call). The statement sequence of a version definition is called the *version body*.

Fig. 33 shows a macro definition with two versions. The definition of the first version, `vers1`, contains all the items mentioned above.

```
t_bmacro(x > y) {
  TEST ?A(x);
  vers1: TEST ?A(y); USE M[m], R[r1,r2]; {
    BRANCH t_b1(x) [s1];
    t_b2(x > m,r1,r2);
    JUMP t_b3() [s2];
    s1: t_b4(x > m,r1,r2);
    s2: t_b5(m,r1,r2 > y);
  }
  vers2: { t_b0(x > y); }
}
```

Figure 33: A macro definition with two versions.

A.5.1 Version-specific test

The version-specific test determines whether the particular macro version can be considered as an expansion-time realization of a call of the current macro. Even if the test succeeds, the version may still be rejected; the complete list of possible rejection reasons is as follows:

- The version-specific test does not succeed.
- Storage cells cannot be allocated for the data variables of the version.
- The macro-specific tests of the macros called in the version body cannot be satisfied.

If all the above three constraints are satisfied, the version is accepted. This decision is irreversible: there is no backtracking during version selection.

For example, consider version `vers1` of macro `t_bmacro` already shown in Fig. 33. The version cannot be selected unless all the following constraints are met:

- Output parameter `y` must represent a cell of storage class `A`.
- The must be a cell of class `M` and two distinct cells of class `R` free.
- The (here unknown) macro-specific tests of lower-level macros `t_b1`, `t_b2`, `t_b3`, `t_b4`, and `t_b5` must be successful.

A.5.2 Temporaries

The set of macro variables may include *macro temporaries*, in addition to macro parameters. Similarly to the macro parameters, the macro temporaries are divided into *data temporaries* and *label temporaries*. Data and label temporaries are valid as cell and label references, respectively, as long as they are not used as form argument designators. Temporaries are version-specific: data temporary is introduced in a version-specific temporary declaration, and a label temporary is introduced by including it as a prefix in some statement in the version body. The temporaries must be different from the parameters and unique within the particular macro version.

For example, consider again version `vers1` in Fig. 33. Its three data temporaries are `m`, `r1`, and `r2` (of storage classes `M`, `R`, and `R`, respectively), and its two label temporaries are `s1` and `s2`.

The macro writer must select the storage class for each data temporary. When the macro expander has chosen the macro version to be linked to a given macro call, it binds each data temporary of the version permanently to some fixed cell of the user-selected class. Data temporaries cannot be made "static"(using the C language terminology): none of them can retain its contents between different execution times of the expansion result.

A.5.3 Initialization of data variables

There are two global constraints on the structure of a single version body; to verify them ReFlEx must perform both data and control flow analysis. These constraints aim to catch mistakes possibly made by the macro writer and are closely related to each other (note that neither one concerns input-output parameters):

- No data temporary or output parameter can be read before it is written.

- Every output parameter must be written.

For example, the body of version `ver` in Fig. 34 would become invalid if the second statement (the call of `t_c2`) were replaced with an empty statement. In that case, in one of the two alternative execution paths,

- temporary `t` is read before it is written,
- output parameter `y` is read before it is written, and
- `y` is not written at all.

```
t_cmacro(x > y,z) {
  ver: USE R[t]; {
    BRANCH t_c1(x) [s1];
    t_c2(x > y,t);
    JUMP t_c3() [s2];
    s1: t_c4(x > y,t);
    s2: t_c5(y,t > z);
  }
}
```

Figure 34: A version body with two alternative execution paths.

The first one of the initialization constraints also enables us, in Appendix B, to formulate a relatively simple definition of a free cell. (In the full ReFlEx 1.0 language, actually, the initialization constraints are less restrictive; see [63, Sec. 6.8].)

A.6 Expansion source

In addition to the rule file, the ReFlEx macro expander needs expansion sources in order to perform code generation. The overall structure of the expansion source is simple:

$$label_{opt} \ \ : \ \ call \ \ ; \ \ disposal_{opt} \ \ follower_{opt}$$

The meaning of the items above is as follows:

label Label associated with the first instruction of the expansion result.

call Macro call to be expanded.

disposal *Disposal set* (see below).

follower *Follower label* (see below).

Thus, the expansion source actually consists of a non-empty statement optionally followed by a disposal set and a follower label. Here are two examples of expansion sources:

```
t_dmacro(M[0] > M[1]);
l0: JUMP t_emacro(R[0]) [l1,l2,l3]; {R[1],R[2],M[100]} >l1
```

Note that in the expansion source there is no need for argument designators: one must explicitly use integer or cell literals instead of data variables (and also code labels instead of label variables—although there is no syntactical difference).

The disposal set contains the storage cells that are free at the point of the macro call—in addition to the output arguments, which are always taken to be free. These cells can be used as a temporary storage inside the expansion result. By default, the disposal set is empty. Above, the disposal set of the second example consists of cells `R[1]`, `R[2]`, and `M[100]`.

The follower label is the label at the end of the expansion result, i.e. the label associated with the instruction that immediately follows the expansion result. In practice, a follower label specification cannot ever be useful unless the follower label is also a label argument of the macro call of the expansion source. (Indeed, the follower label `l1` of the second example above is also a label argument of the `t_emacro` call.)

B ReFlEx 1.0 expansion mechanism

The ReFlEx 1.0 macro expansion begins from an *expansion source* and produces an *expansion result*. The expansion source consists of a single macro call, and the expansion result is a sequence of macro calls. In this section we give a precise description of the macro expansion procedure.

B.1 Phases of the procedure

During the macro expansion a tree structure is built upon the source macro call by repeated refinement steps. Each *node* of this *expansion tree* is either an empty statement or a macro call statement; one of the nodes is the *root*, which contains the source macro call. The nodes of the tree can be divided into *stock nodes* and *leaf nodes*; each node with an empty statement is a leaf (thus, an empty statement is actually equivalent to a call of an atomic 'no-op' macro). To the macro call of each stock node a *realization* has already been linked, while the macro calls of the leaf nodes have not yet been provided with a realization. The realization of a macro call is selected from the versions of the called macro; the lower-level macro calls in the body of the selected version become new leaf nodes. Once each non-empty leaf of the tree is a call of an atomic macro, there is no need for further refinement of the tree. Then the tree is ready to linearized into a macro call sequence consisting of the final set of leaf nodes.

Thus, as shown even in Fig. 35, the macro expansion procedure comprises three successive phases:

1. Setup phase.
2. Linking phase.
3. Merging phase.

At the beginning of the expansion, the *setup phase* sets up an initial expansion tree. The initial tree consists of a single root node which contains the macro call of the expansion source. The disposal set and the possible follower label from the expansion source are also associated with the tree.

The *linking phase* is the essential component of the macro expansion procedure. It builds up the hierarchical expansion tree by repeatedly augmenting the structure originating from the initial expansion tree. A more detailed description of the linking phase will be given in Sec. B.2.

The *merging phase* mechanically linearizes the complete expansion tree into an expansion result: the macro calls of the leaf nodes in the expansion tree are

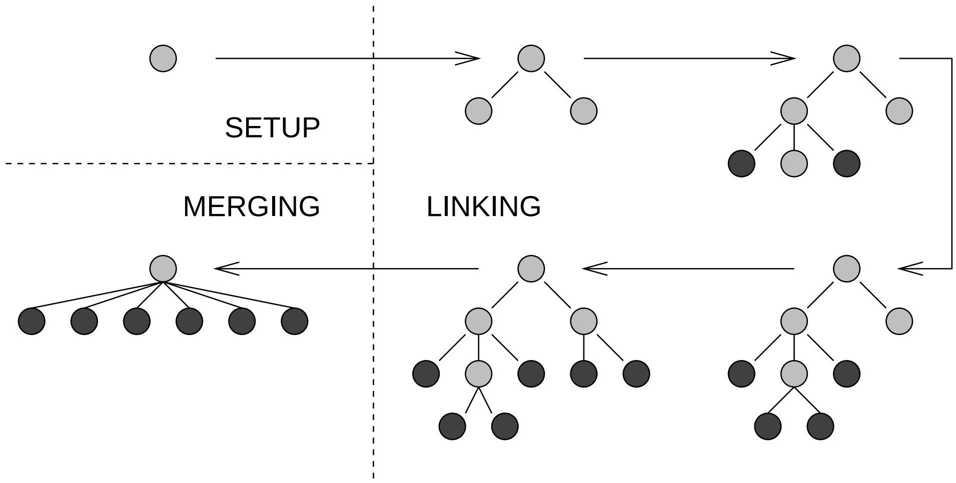

Figure 35: Phases of the macro expansion procedure.

collected into a flat macro call sequence. This linearization means that the natural sequential order of the leaf nodes is preserved while all the following nodes are dropped off from the tree:

- All the stock nodes.
- Each unreachable leaf node.

Unreachable leaf nodes may exist, because macro definitions may well include versions that do not refer to every exit parameter.

Example B.1
Sec. 4.3.5 begins with a specification of a sample expansion source. The full expansion tree and the final flat expansion result produced by ReFlEx are shown in Fig. 17 on page 35. □

B.2 Linking phase

The linking phase comprises a number of successive steps: each step links a realization to a macro call (more specifically, to a call of a composite macro), which thus changes from a leaf node into a stock node of the expansion tree. Each step includes two selections:

1. A leaf to be processed next has to be selected.
2. A suitable realization version for the selected leaf macro call must be selected from the versions of the particular macro.

Each statement in the body of the selected version constitutes a new leaf node in the tree. Leaf selection is dealt with in Sec. B.2.1, and an overview of version selection is given in Sec. B.2.2. The following Sec. B.3 will complete the description of version selection.

The success or failure of the macro expansion can be determined already during the linking phase.

Definition B.1 *An expansion tree is a **failure** if it meets at least one of the following conditions:*
(a) *The macro-specific test of the root macro call is not successful.*
(b) *The tree contains a composite macro call leaf to which no realization version can be linked.*

If the macro-specific test of the root macro call is not successful, ReFlEx does not even try to link additional nodes to the tree. (If the tree can be built, however, the macro-specific tests of the additional nodes are checked as a part of the version selection; see Sec. B.2.2.)

It is perfectly possible that a composite macro call leaf for which no suitable realization version exists is added to the tree. This means an irreversible failure, because backtracking is not supported. Still, in this case ReFlEx does not stop the linking phase as long as there also exists such leaves in the tree for which a suitable version can be found.

Definition B.2 *An expansion tree is **complete** if its each leaf meets at least one of the following conditions:*
(a) *The leaf is the root and consists of a macro call that does not meet its macro-specific test.*
(b) *The leaf consists of an empty statement.*
(c) *The leaf consists of an atomic macro call.*
(d) *The leaf consists of a composite macro call for which there is no acceptable macro version.*

Again, the *acceptability* of a macro version will be defined in Sec. B.2.2, which deals with version selection in detail.

Recursive macro definitions are possible. In practice, however, the linking phase is guaranteed to end, since infinite recursion is disabled: the ReFlEx user must set a limit on the number of nodes in the expansion tree (and, correspondingly, on the allowed form call recursion depth).

B.2.1 Leaf selection

How should the leaf node to be processed next be selected (of course, this question is relevant only if the already generated expansion tree has at least two leafs that consist of calls of composite macros)? The answer is that such a selection, which is depicted in Fig. 36, is fully insignificant. The reason for this is that once a leaf node is inserted in the expansion tree, the set of cells that are free at it never changes. This fact becomes explicit in Sec. B.3.2 below (and is thoroughly discussed in Sec. 5).

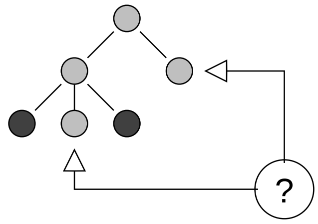

Figure 36: Leaf selection.

Actually, the linking and merging phases are separated just in order to achieve the insignificance of leaf selection. If the results of individual linking steps were merged into the expansion result under construction right away, the free cell sets would not always remain unchanged. Moreover, such right-away merging could only make the free cell sets larger, which may at first seem desirable: the refinement of a particular "difficult" leaf could then be delayed unless the current number of free cells were large enough. This enlargement of free cell sets through right-away merging could, however, happen only in some exceptional cases; see Ex. B.2 for an instance of such an exception.

Example B.2
Here is an expansion tree with two leaves, i.e. calls of `t_ord1` and `t_ord2`:

```
t_order(A,R[2] > A,R[2])  { }
      t_ord1(A > A)
      t_ord2(A,R[2] > R[2])
```

Clearly, register `R[2]` is not free at the `t_ord1` call, as `R[2]` is an input argument of the original `t_order` call and read by the `t_ord2` call (the actual definition of freedom can be found in Sec. B.3.2). But suppose that the definition of `t_ord2` is as follows (note in particular that input parameter `b` is not read in the single version of `t_ord2`):

```
t_ord2(a,b > c) {
  impl: { t_ord3(a > c); }
}
```

Let us then link the single `t_ord2` version to the `t_ord2` leaf. If we now broke the rules and merged the version body immediately into the tree, we would get the following:

```
t_order(A,R[2] > A,R[2])  { }
      t_ord1(A > A)
      t_ord3(A > R[2])
```

Notice that R[2] would then become free at the t_ord1 call. □

For the advantages offered by the insignificance of leaf selection, see Sec. 3.4.3.

B.2.2 Version selection

Once a leaf node is selected, a suitable realization version for its macro call must be selected from the versions of the particular macro. ReFlEx traverses the versions shown in the rule file in the order specified by the macro writer. This traversal ends as soon as an *acceptable* version is found; if no version is acceptable, the whole expansion of the current expansion source fails.

Definition B.3 *A particular macro version is* ***acceptable*** *to a particular leaf if both the following conditions are met:*
(a) *The version-specific test is successful.*
(b) *There exists a macro variable binding that meets both the following sub-conditions:*

- *It is legitimate.*
- *It makes the macro-specific test of each macro call in the version body successful.*

The *macro variable binding* specifies the integers, cells, and labels that substitute for the data and label temporaries of the macro version. The *legitimacy* of a variable binding depends on the context (e.g. the set of cells that are free at the leaf) and will be formulated in Def. B.7 in Sec. B.3.3. Finally, to evaluate the lower-level macro-specific tests, ReFlEx has to create a provisional link between the leaf node and an appropriately modified copy of the version body.

Example B.3
Here is a simple macro definition with a single version, i.e. impl:

```
t_accept(x > y) {
  TEST %(x,y);
  impl: TEST ?R(x); USE R[t]; {
    t_acc1(x > t); t_acc2(x,t > y);
  }
}
```

Suppose now that the expander is about to link a realization to a t_accept leaf in some expansion tree. What are the conditions on the acceptability of the impl version now like?

The version-specific test requires that the input argument corresponding to parameter `x` must be a cell of storage class `R`. Obviously, there cannot be a sound variable binding unless there is a cell of class `R` to which macro temporary `t` can be bound and that is both free at the leaf and different from the input argument. Furthermore, each one of macros `t_acc1` and `t_acc2` may have a macro-specific test that must be evaluated. The existence of macro-specific tests can be determined only by looking at the respective macro definitions. For instance, `t_accept` here has a macro-specific test that requires its two arguments to belong to the same storage class (this constraint has been verified already before the insertion of the `t_accept` call in the tree as a leaf). If `t_acc1`, say, happened to have an exactly similar test, then that test would be guaranteed to succeed, since both `x` and `t` are associated with class `R`. (In general, of course, the fate of the macro-specific test of a given macro call inside a given version definition is not fixed orthostatically, that is, before the actual expansion. The result might depend, say, on the number of cells free at the macro call, in which case the provisional link mentioned above is usually needed.) □

If a particular macro version is accepted, an appropriately modified copy of the version body is linked to the leaf node and thus irreversibly attached to the expansion tree. Once the version selection is made, it cannot ever be canceled, not even in the case that a blind alley is later met—backtracking in version selection is not supported. Therefore, the ordering of the macro versions and the contents of their version-specific tests should be orchestrated very carefully.

B.3 Conditions on variable binding

In this section we derive the crucial part of the linking phase description: the definition of a legitimate variable binding. We must begin by introducing some preliminary notions.

B.3.1 Preliminary notions

We aim at generality and brevity in our discussion; in particular, we want that some crucial notions apply to both macro definitions and expansion trees. We begin with some simple generalizing conventions:

- Cells and data variables are collectively called *markers*.
- We assume that ReFlEx quietly associates a unique *implicit label temporary* with each such statement in a macro version body

that the user has not explicitly provided with a label temporary; moreover, these implicit label temporaries are indistinguishable from the original ones during macro expansion.

- The macro head is a syntactically valid macro call; since we want to be able to treat it as a statement, we assume that ReFlEx provides even it with an implicit label temporary.
- If the user has not explicitly provided the statement within the expansion source with a label, ReFlEx provides it with an arbitrary label that does not already appear in the expansion source.

Because of the implicit label temporaries, no macro version body can contain two identical statements. Furthermore, as indicated by Def. B.7 in Sec. B.3.3, even an expansion tree cannot contain two identical statements. (Here we have a discrepancy between our current terminology and the one used in Sec. 5. In Sec. 5, each macro call contains a label as an integral part; here the tree nodes are constituted by statements consisting of a macro call and a separate label. The latter convention is perhaps more intuitive, whereas the former one was found to be more economical from the formalization viewpoint.)

Next, we give our main umbrella definition.

Definition B.4 *A **macro scope** or, more shortly, a **scope** consists of a **scope head**, which is a statement, and a **scope body**, which is a sequence of statements.*

Intuitively, the scope body is an implementation of the scope head. We introduce two scope types, *definition scopes* and *expansion scopes*:

- For a macro definition, there are as many definition scopes as there are macro versions. The head of each definition scope is the macro head statement, and the scope body is the body of one of the macro versions.
- For an expansion tree, there are as many expansion scopes as there are stock nodes. The head of each expansion scope is the macro call statement of the stock node, and the scope body is the realization of that macro call. Thus, any stock node statement other than the root statement both is a head of an expansion scope and belongs to the body of another expansion scope.

Example B.4
Fig. 37 shows a macro definition with two definitions scopes; Fig. 38 shows an expansion tree with three expansion scopes. In Fig. 38, the root node is provided with a non-empty free cell set—this free cell set must be explicitly determined in the expansion source by the user. □

```
BRANCH tmpA(x,y > z,w) [l] {
  v1: {
    BRANCH tmpB(x,y > z,w) [l];
  }
  v2: USE R[t]; {
    tmpC(x > t);
    tmpD(y > z);
    l0: tmpE(t > t);
    BRANCH tmpF(z > y,w) [l];
    tmpG(y > t);
    tmpH(y > z);
    BRANCH tmpI(t) [l0];
    tmpJ(t > w);
  }
}
```

Figure 37: A macro definition with two definition scopes.

```
tmp0(M[1] > M[2])  {A,R[0],R[3],M[1],M[2]}
      tmp1(M[1] > M[2],R[3])
            tmp11(M[1] > M[2],R[3])
      tmp2(M[1] > A)
            tmp21(M[1] > R[0])
            tmp22(M[1] > A)
            tmp23(A,R[0] > A)
      tmp3(R[3],M[2] > R[3])
      tmp4(R[3],A > M[2])
```

Figure 38: An expansion tree with three expansion scopes.

We also introduce some notions related to execution-time control propagation. First, concerning intrascope control propagation, we adopt the following terminology:

- Each statement in a scope body has a set of *immediate successor* statements associated with it. For instance, a two-way branch typically has two immediate successors.
- A *path* in a scope body is a chain of statements that is glued together by the immediate successorship.

Second, two more terms concern interscope control propagation:

- Each scope body has exactly one *entry point*, which is the first statement in the scope body, i.e. the statement that receives the control from the outside.
- Each scope body has a set of *exit points*: any statement in the scope body that may relinquish the control outside the scope is

an exit point. (The last statement in the scope body is an exit point unless the deviation qualifier of its macro call is `JUMP` or `SLEEP`; additionally, any statement whose macro call has an exit argument in common with the scope head is an exit point.)

Example B.5
Look again at Fig. 37 on page 78. The following sequence is a path from an entry point of a scope to an exit point:

C→D→E→F→G→H→I→E→F

(For brevity, we have here dropped out the '`tmp`' prefixes: '`A`' stands for '`tmpA`', and so on. This convention holds even for a couple of following examples.) □

B.3.2 Free cell analysis

In this subsection, we formulate a precise definition for the notion of a free marker, that is, a free cell or a free data variable. We begin by defining life paths, which indicate data dependencies that span over statements in the code.

Definition B.5 *A path is a* ***life path*** *of a marker if all the following conditions are met:*
(a) *No statement in the path writes the marker.*
(b) *The first statement of the path meets at least one of the following subconditions:*

- *It is an immediate successor of some statement that writes the marker.*
- *It is an entry point of the scope body, and the scope head reads the marker.*

(c) *The last statement of the path meets at least one of the following subconditions:*

- *It has an immediate successor that reads the marker.*
- *It is an exit point of the scope body, and the scope head writes the marker.*

Example B.6
Consider again Fig. 37 on page 78. For the macro variables referred to in the `v2` version, there are the following life paths:

```
x :   –
y :   C  G
z :   E  I→J  I→E  E→F  I→E→F
w :   –
t :   D  H  H→I
```

□

Finally, we are able to define the set of free markers for each statement in any scope, that is, the set of free variables for each statement in any definition scope and the set of free cells for each statement in any expansion scope.

Definition B.6 *Whether a marker is* ***free*** *at a statement in a macro scope, is determined according to the following criteria:*
(a) *Every data variable is free at the definition scope head.*
(b) *A cell is free at the root statement of an expansion tree if the cell either belongs to the disposal set of the expansion tree or is written by the root statement.*
(c) *A marker is free at a statement in a scope body if both the following subconditions are met:*

- *The statement belongs to no life path of the marker.*
- *The marker is free at the scope head.*

Note in particular that in the case of expansion scopes the definition above is recursive. Moreover, the freedom in expansion scopes depends only on the structure of the current scope and the enclosing upper-level scopes, and on the disposal set. Effectively, this means that the order in which the appropriate realizations are linked to the leaves of the expansion tree is indeed insignificant, as we claimed already in Sec. B.2.1. (Often we simply say that a cell or a data variable is free at some macro call, even if we actually mean freedom at the statement that contains the macro call.)

Example B.7
Consider still one more time Fig. 37 on page 78. The macro variables of version `v2` are, according to Ex. B.6, *not* free at the following statements:

```
x :   –
y :   C  G
z :   E  F  I  J
w :   –
t :   D  H  I
```

□

Example B.8
Consider now Fig. 38 on page 78. The cells free at the root are *not* free at the following non-root nodes:

```
A    :   3
R[0] :   22
R[3] :   2  21  22  23
M[1] :   1  11  21
M[2] :   2  21  22  23
```

□

B.3.3 Legitimacy

When a macro version body is about to be linked to a leaf macro call, each macro variable of the version is mapped to a single integer, cell, or label. This *variable binding* must respect a number of constraints, which we now specify.

Definition B.7 *A variable binding is* ***legitimate*** *if all the following conditions are met:*
(a) *Each macro parameter must be bound to the integer, cell, or label that is the corresponding argument of the leaf macro call: an input parameter to an integer or to a cell, an output or input-output one to a cell, and an exit one to a label.*
(b) *Each data temporary must be bound to a cell of the indicated storage class.*
(c) *Each data variable (each input parameter included) that is written by some macro call in the version body must be bound to a cell that is free at the leaf macro call.*
(d) *If two distinct data variables are bound to a single common cell, and if one of them is written by some macro call in the version body, then the other one must not be written by that macro call.*
(e) *If two distinct data variables are bound to a single common cell, and if one of them is written by some macro call in the version body, then the other one must be free at that macro call.*
(f) *Each label temporary must be bound to a label that does not already appear in the expansion tree, that is different from the possible follower label of the tree, and to which no other label temporary is bound.*

From Def. B.7 we easily see that a legitimate label temporary binding can always be found, since we can assume that the set of labels is countably infinite. Furthermore, all legitimate label temporary bindings are trivially equivalent. (For data temporaries, in contrast, see Sec. B.4.)

Example B.9
Suppose that we have a root node

```
L3: mac0(M[1] > M[2])  {R[0],R[3]}
```

and a single-version macro definition,

```
mac0(m1 > m2) {
  v0: USE R[r]; {
    mac1(m1 > r,m2);
    l0: mac2(r,m2 > m2);
    BRANCH mac3(m1,m2 > m2) [l0,l0];
  }
}
```

which we want to link to the root node specified in the expansion source.

For simplicity, we assume that each macro dealt with in this example contains neither macro-specific nor version-specific tests; this means that a legitimate variable binding is a sufficient condition for version acceptability (see Def. B.3). By this assumption, the linking is clearly possible, because there is a free cell of class `R` to be overitten [condition (c) of Def. B.7], i.e. `R[0]` or `R[3]`. Thus we get an expansion tree such as shown in Fig. 39.

```
L3: mac0(M[1] > M[2])  {R[0],R[3]}
      mac1(M[1] > R[3],M[2])
      L5: mac2(R[3],M[2] > M[2])
      BRANCH mac3(M[1],M[2] > M[2]) [L5,L5]
```

Figure 39: The tree after the first link operation.

Suppose then that we want to link a realization to the `mac3` node in the tree. The definition of `mac3` is shown in Fig. 40. The argument list of the `mac3` call is indeed compatible with the parameter list of the `mac3` definition: the two exit arguments of the `mac3` call are identical, as required by the macro head [condition (a)].

```
BRANCH mac3(m1,m2 > m3) [l,l] {
  v1: {
    macA(m1,m2 > m1,m3);
    BRANCH macB(m1,m3 > m3) [l];
  }
  v2: {
    macA(m1,m2 > m2,m3);
    BRANCH macB(m2,m3 > m3) [l];
  }
  v3: USE M[mx,my]; {
    macC(m1,m2 > mx);
    macD(m1,mx > my);
    BRANCH macB(mx, my > m3) [l];
  }
  v4: USE R[r], M[m]; {
    macC(m1,m2 > m);
    macD(m1,m > r);
    BRANCH macB(m,r > m3) [l];
  }
}
```

Figure 40: A macro definition with four versions.

We traverse the versions of `mac3` in the order given:

- version v1 must be rejected, because it writes input parameter m1, but the corresponding argument, i.e. M[1], is not free [condition (c)].
- v2 must be rejected, because variables m2 and m3 must be bound to a common cell (as the arguments standing for these two parameters are identical), but the macA call in the version body writes both the variables [condition (d)].
- v3 must be rejected, because both mx and my must not be bound to the only free cell of class M, i.e. M[2]: mx is not free at the macD call, by which my is written [condition (e)].

How about the final v4 then? You might verify by yourself that there is a legitimate variable binding for v4; the result of the second linking can be seen in Fig. 41.

```
L3: mac0(M[1] > M[2])  {R[0],R[3]}
      mac1(M[1] > R[3],M[2])
      L5: mac2(R[3],M[2] > M[2])
      BRANCH mac3(M[1],M[2] > M[2]) [L5,L5]
            macC(M[1],M[2] > M[2])
            macD(M[1],M[2] > R[0])
            BRANCH macB(M[2],R[0] > M[2]) [L5]
```

Figure 41: The tree after the second link operation.

□

B.4 On intraclass cell allocation

For a macro version that is about to be linked to a leaf node of an expansion tree, there may be several legitimate but still essentially different data temporary bindings available. For instance, there might be two possible alternatives for a pair of data temporaries of a single common storage class: either the temporaries are bound to a single common cell or they are bound to two separate cells. We intentionally leave it unspecified how ReFlEx chooses among alternative legitimate data temporary bindings. Still, a wrong choice may later prove to be a fatal mistake: the whole expansion may result in a failure that could have been prevented by a better choice.

In practice, however, the user can prevent unwanted data temporary bindings. With the rudimentary ReFlEx 1.0, the only significant difference between two legitimate data temporary bindings can be reduced to a set of choices of the following kind: are some argument designators of some individual macro call in the version body bound to a single cell or to distinct cells? (Accordingly, no diagnostic primitive form can differentiate between legitimate choices of other kinds.) By suitably augmenting the macro-specific tests of the appropriate lower-level macros, the user can therefore force the desired binding. (Remember that ReFlEx rejects even a legitimate variable binding if the macro-specific tests of the lower-level macro calls are not satisfied, as stated in Def. B.3 in Sec. B.2.2).

Example B.10
Suppose that we have the following macro definitions:

```
t_weird(a > r) {
  wver: USE R[t]; { my_move(a > t); t_clash(t > r); }
}
```

Furthermore, the definition of `t_clash` (which is called inside `t_weird` above) is supposed to be as follows:

```
t_clash(t > r) {    // suspicious?
  cver: TEST =(t,r); { my_move(t > r); }
}
```

Let us then assume that we are faced with the following expansion source:

```
t_weird(A > R[0]); {R[1]}
```

Thus we have two free cells of storage class `R`, i.e. `R[0]` and `R[1]`. Temporary `t` of `t_weird` must clearly be bound to either one of these. If `R[0]` is selected, everything goes well, as the argument designators of the lower-level `t_clash` call inside the `wver` version body both become bound to `R[0]`. But if `R[1]`, instead of `R[0]`, is selected, then the `t_clash` call cannot be provided with a realization: the only version of `t_clash` expects that the two argument designators in the call are bound to the same cell. The macro writer should avoid this pitfall by moving the crucial version-specific test of the `cver` version into a macro-specific test of the whole `t_clash` macro; here we have the improved definition:

```
t_clash(t > r) {    // an improvement!
  TEST =(t,r);
  cver: { my_move(t > r); }
}
```

□

Accordingly, as only some of the intraclass allocation decisions are significant, it is readily seen that the automatic allocation performed by ReFlEx 1.0 can be optimized as follows. When the built-in allocator is traversing through all the legitimate bindings in order to find one which could make the version acceptable, many of the possible bindings can be skipped right away, that is, without evaluation of all the lower-level macro-specific tests. More precisely, any given binding N can be skipped if there is another binding M that has already been found to be wanting, and no macro call in the macro version body *properly differentiates* between M and N. Here we say that a macro call properly differentiates between two variable bindings if the call has two data argument designators which one of the bindings maps to a single cell but which the other one maps to two distinct cells.

C ReFlEx 1.0 code generation examples

C.1 A rule file

Here we present an example of a syntactically valid ReFlEx 1.0 rule file. The rule file is an extension of the one dealt with in Sec. 4.

```
HEADER {
  Fact(a) = _If(_Lt(a,2), 1, _Mul(a,Fact(_Add(a,-1))));
  Add(a,b) = _Add(a,b);
  BShl(a,b) = _BShl(a,b);
  Not(a) = _Nand(a,a);
  And(a,b) = _If(a, _If(b,1,0), 0);
  Or(a,b) = _If(a, 1, _If(b,1,0));
  Int(a) = And(Not(?(a)), Not(&(a)));
  Lt(a,b) = And(_Lt(a,b), And(Int(a),Int(b)));
  Gte(a,b) = And(Not(Lt(a,b)), And(Int(a),Int(b)));
  Eq(a,b) = And(Gte(a,b), Gte(b,a));
}

STORAGE {
  M[1024];
  R[4];
  A[1];
}

SYSTEM {
  set(c > r) { TEST And(?R(r), And(Gte(c,-1024),Lt(c,1024))); }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) { TEST And(Or(?A(s),?R(s)), Or(?A(d),?R(d))); }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}

UTILITY {

my_null(x > x) {
    // verifies that the arguments are the same
  null: { }
}

my_move(s > d) {
    // moves s into d
```

```
  same: { my_null(s > d); }
  as_set: { set(s > d); }
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  temp_needed: TEST And(Not(?R(s)), Not(?R(d))); USE R[r];
    { my_move(s > r); my_move(r > d); }
}

my_mswap(m,n > n,m) {
    // exchanges the contents of two memory locations
  TEST ?M(m,n);
  two_aux_free: TEST Gte(#R(),2); USE R[r];
    { my_move(m > r); my_move(n > m); my_move(r > n); }
  acc_and_aux_free: USE A[a];
    { my_move(m > a); my_move(n > m); my_move(a > n); }
}

BRANCH my_gt(x) [l] {
    // branches if x is positive
  next: TEST &(l,NEXT); { }
  const0: TEST Gte(x,1); { JUMP goto() [l]; }
  const: TEST Int(x); { }
  acc: TEST ?A(x); { BRANCH gt(x) [l]; }
  default: USE A[a]; { my_move(x > a); BRANCH gt(a) [l]; }
}

BRANCH my_gt_decr(x,d > y) [l] {
    // y is x decreased by d;
    // branches if y remains positive
  x_not_acc: TEST Not(?A(x)); USE A[a]; {
    my_move(x > a);
    BRANCH my_gt_decr(a,d > y) [l];
  }
  d_not_aux: TEST Not(?R(d)); USE R[r]; {
    my_move(d > r);
    BRANCH my_gt_decr(x,r > y) [l];
  }
  y_not_acc: TEST Not(?A(y)); USE A[a]; {
    sub(x,d > a);
    my_move(a > y);
    BRANCH my_gt(a) [l];
  }
  default: {
    sub(x,d > y);
    BRANCH my_gt(y) [l];
  }
}
```

```
my_fix_lshift(x,s > y) {
    // y is x shifted left by s bit positions;
    // s must be a non-negative integer constant
  TEST And(Int(s),Gte(s,0));
  src_const: TEST Int(x); { my_move(BShl(x,s) > y); }
  zero_shift: TEST Eq(s,0); { my_move(x > y); }
  dst_not_acc: TEST Not(?A(y)); USE A[a]; {
    my_fix_lshift(x,s > a); my_move(a > y);
  }
  src_mem: TEST ?M(x); USE R[r]; {
    my_move(x > r); my_fix_lshift(r,s > y);
  }
  src_aux: TEST ?R(x); {
    my_move(x > y); add(y,x > y); my_fix_lshift(y,Add(s,-1) > y);
  }
  src_acc: USE R[r]; {
    my_move(x > r); add(x,r > y); my_fix_lshift(y,Add(s,-1) > y);
  }
}

my_var_lshift(x,s > x) {
    // x shifted left by s bit positions;
    // s must be in the accumulator and
    // x in an auxiliary register
  TEST And(?R(x),?A(s));
  implem: USE R[d,t]; {
    BRANCH eq(s) [l2];
    my_move(1 > d);
    l1: my_move(s > t);
    my_fix_lshift(x,1 > x);
    my_move(t > s);
    BRANCH my_gt_decr(s,d > s) [l1];
    l2: ;
  }
}

}
```

C.2 Example runs

Here we present some code generation examples produced with the rule file of Sec. C.1. Before the final expansion result, we each time show the full expansion tree. Note again that the actual ReFlEx 1.0 output syntax is somewhat more complicated (see [63] for a complete reference).

```
> my_mswap(M[8],M[6] > M[6],M[8]) {R[1],R[3]}
      my_move(M[8] > R[3])
```

```
        load(M[8] > R[3])
    my_move(M[6] > M[8])
        my_move(M[6] > R[1])
            load(M[6] > R[1])
        my_move(R[1] > M[8])
            store(R[1] > M[8])
    my_move(R[3] > M[6])
        store(R[3] > M[6])

  my_mswap(M[8],M[6] > M[6],M[8]) {R[1],R[3]}
      load(M[8] > R[3])
      load(M[6] > R[1])
      store(R[1] > M[8])
      store(R[3] > M[6])

> my_mswap(M[8],M[6] > M[6],M[8]) {A,R[3]}
      my_move(M[8] > A)
          my_move(M[8] > R[3])
              load(M[8] > R[3])
          my_move(R[3] > A)
              move(R[3] > A)
      my_move(M[6] > M[8])
          my_move(M[6] > R[3])
              load(M[6] > R[3])
          my_move(R[3] > M[8])
              store(R[3] > M[8])
      my_move(A > M[6])
          my_move(A > R[3])
              move(A > R[3])
          my_move(R[3] > M[6])
              store(R[3] > M[6])

  my_mswap(M[8],M[6] > M[6],M[8]) {A,R[3]}
      load(M[8] > R[3])
      move(R[3] > A)
      load(M[6] > R[3])
      store(R[3] > M[8])
      move(A > R[3])
      store(R[3] > M[6])

> my_fix_lshift(M[10],3 > M[11]) {A,R[3]}
      my_fix_lshift(M[10],3 > A)
          my_move(M[10] > R[3])
              load(M[10] > R[3])
          my_fix_lshift(R[3],3 > A)
              my_move(R[3] > A)
                  move(R[3] > A)
              add(A,R[3] > A)
              my_fix_lshift(A,2 > A)
```

```
                my_move(A > R[3])
                    move(A > R[3])
                add(A,R[3] > A)
                my_fix_lshift(A,1 > A)
                    my_move(A > R[3])
                        move(A > R[3])
                    add(A,R[3] > A)
                    my_fix_lshift(A,0 > A)
                        my_move(A > A)
                            my_null(A > A)
    my_move(A > M[11])
        my_move(A > R[3])
            move(A > R[3])
        my_move(R[3] > M[11])
            store(R[3] > M[11])

  my_fix_lshift(M[10],3 > M[11]) {A,R[3]}
    load(M[10] > R[3])
    move(R[3] > A)
    add(A,R[3] > A)
    move(A > R[3])
    add(A,R[3] > A)
    move(A > R[3])
    add(A,R[3] > A)
    move(A > R[3])
    store(R[3] > M[11])

> my_var_lshift(R[0],A > R[0]) {A,R[2],R[3]}
    BRANCH eq(A) [l2$1]
    my_move(1 > R[2])
        set(1 > R[2])
    l1$1: my_move(A > R[3])
        move(A > R[3])
    my_fix_lshift(R[0],1 > R[0])
        my_fix_lshift(R[0],1 > A)
            my_move(R[0] > A)
                move(R[0] > A)
            add(A,R[0] > A)
            my_fix_lshift(A,0 > A)
                my_move(A > A)
                    my_null(A > A)
        my_move(A > R[0])
            move(A > R[0])
    my_move(R[3] > A)
        move(R[3] > A)
    BRANCH my_gt_decr(A,R[2] > A) [l1$1]
        sub(A,R[2] > A)
        BRANCH my_gt(A) [l1$1]
            BRANCH gt(A) [l1$1]
```

```
    l2$1:

my_var_lshift(R[0],A > R[0]) {A,R[2],R[3]}
    BRANCH eq(A) [l2$1]
    set(1 > R[2])
    l1$1:
    move(A > R[3])
    move(R[0] > A)
    add(A,R[0] > A)
    move(A > R[0])
    move(R[3] > A)
    sub(A,R[2] > A)
    BRANCH gt(A) [l1$1]
    l2$1:
```

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 4, pp. 491–516. October 1989.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading (Massachusetts, USA), 1986.

[3] F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H. Harrison, P. G. Loewner, and L. H. Trevillyan. The experimental compiler system. *IBM Journal of Research and Development*, vol. 24, no. 6, pp. 695–715. November 1980.

[4] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sundarsanam, S. Tjiang, and A. Wang. Challenges in code generation for embedded processors. In [74], pp. 48–64.

[5] B. W. Arden, B. A. Galler, and R. M. Graham. The MAD definition facility. *Communications of the ACM*, vol. 12, no. 8, pp. 432–439. August 1969.

[6] D. Arnold, L. Balkan, R. Lee Humphreys, S. Meijer, and L. Sadler. *Machine Translation: An Introductory Guide.* NCC Blackwell, Oxford (UK), 1994.

[7] AT&T Microelectronics. DSP1610 digital signal processor information manual (MN90-020DMOS). Allentown (Pennsylvania, USA), December 1992.

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. Mitchell Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. *Proceedings of 1957 Western Joint Computer Conference*, pp. 188–197. 1957.

[9] J. W. Backus and W. P. Heising. FORTRAN. *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 4, pp. 382–385. August 1964.

[10] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420. December 1994.

[11] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Journal of Computer Languages*, vol. 15, no. 3, pp. 127–140. 1990.

[12] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam (The Netherlands), 1981.

[13] C. Bass. PLZ: a family of system programming languages for microprocessors. *IEEE Computer*, vol. 11, no. 3, pp. 34–39. March 1978.

[14] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In J. Gutknecht (ed.), *Programming Languages and System Architectures*, pp. 105–128. Springer, Berlin (Germany), 1994.

[15] P. J. Brown. A survey of macro processors. *Annual Review in Automatic Programming*, vol. 6, part 2, pp. 37–88. Pergamon Press, Oxford (UK), 1969.

[16] P. J. Brown. *Macro Processors and Techniques for Portable Software*. Wiley, London (UK), 1974.

[17] M. Campbell-Kelly. *An Introduction to Macros*. Macdonald, London (UK), 1973.

[18] T. E. Cheatham. The introduction of definitional facilities into higher level programming languages. *Proceedings of AFIPS 1966 Fall Joint Computer Conference*, pp. 623–637. 1966.

[19] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, vo. 12, no. 4, pp. 501–536. October 1990.

[20] A. J. Cole. *Macro Processors* (second edition). Cambridge University Press, Cambridge (UK), 1981.

[21] S. Crespi-Reghizzi, P. Corti, and A. Dapra'. A survey of microprocessor languages. *IEEE Computer*, vol. 13, no. 1, pp. 48–66. January 1980.

[22] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London (UK), 1972.

[23] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 505–526. October 1984.

[24] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, vol. 15, no. 2, pp. 26–41. February 1982.

[25] D. Desmet and D. Genin. ASSYNT: efficient assembly code generation for digital signal processors starting from a data flowgraph. *Proceedings of 1993 International Conference on Acoustics, Speech, and Signal Processing*, vol. III, pp. 45–48. IEEE, 1993.

[26] B. N. Dickman. ETC—an extendible macro-based compiler. *Proceedings of AFIPS 1971 Spring Joint Computer Conference*, pp. 529–538. 1971.

[27] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, vol. 11, no. 3, pp. 147–148. March 1968.

[28] M. Elson and S. T. Rake. Code-generation technique for large-language compilers. *IBM Systems Journal*, vol. 9, no. 3, pp. 166-188. 1970.

[29] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, vol. 29, no. 6 (Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation), pp. 242–256. June 1994.

[30] H. Emmelmann, F.-W. Schöer, and R. Landwehr. BEG—a generator for efficient back ends. *ACM SIGPLAN Notices*, vol. 24, no. 7 (Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation), pp. 227–237. July 1989.

[31] D. J. Farber. A survey of the systematic use of macros in systems building. *ACM SIGPLAN Notices*, vol. 6, no. 9 (Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation), pp. 29–36. October 1971.

[32] E. Farquhar and P. Bunce. *The MIPS Programmer's Handbook*. Morgan Kaufmann, San Francisco (California, USA), 1994.

[33] E. D. Ferguson. Evolution of the meta-assembly program. *Communication of the ACM*, vol. 9, no. 3, pp. 190–196. March 1966.

[34] C. N. Fischer and R. J. LeBlanc. *Crafting A Compiler*. Benjamin/Cummings, Menlo Park (California, USA), 1988.

[35] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490. July 1981.

[36] J. A. Fisher and B. R. Rau. Instruction-level parallel processing. *Science*, vol. 253, no. 5025, pp. 1233–1241. September 1991.

[37] C. W. Fraser. A language for writing code generators. *ACM SIGPLAN Notices*, vol. 24, no. 7 (Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation), pp. 238–245. July 1989.

[38] C. W. Fraser, D. R. Hanson, and T. A. Proebstring. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, pp. 213–226. September 1992.

[39] M. Ganapathi and C. N. Fisher. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 560–599. October 1985.

[40] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. Retargetable Compiler Code Generation. *ACM Computing Surveys*, vol. 14, no. 4, pp. 573–592. December 1982.

[41] J. G. Ganssle. *The Art of Programming Embedded Systems.* Academic Press, San Diego (California, USA), 1992.

[42] R. S. Glanville. A machine-independent algorithm for code generation and its use in retargetable compilers. Ph. D. thesis, University of California. Berkeley (California, USA), December 1977.

[43] R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conference Record of 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240. January 1978.

[44] S. L. Graham. Table-driven code generation. *IEEE Computer*, vol. 13, no. 8, pp. 25–34. August 1980.

[45] I. D. Greenwald. A technique for handling macro instructions. *Communications of the ACM*, vol. 2, no. 11, pp. 21–22. November 1959.

[46] R. E. Griswold. *The Macro Implementation of SNOBOL4.* W. H. Freeman, San Francisco (California, USA), 1972.

[47] M. L. Halpern. Toward a general processor for programming languages. *Communications of the ACM*, vol. 11, no. 1, pp. 15–25. January 1968.

[48] W. H. Harrison. A new strategy for code generation—the general-purpose optimizing compiler. *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, pp. 367–373. July 1979.

[49] M. S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York (New York, USA), 1977.

[50] R. R. Henry. Graham-Glanville code generators. Ph. D. thesis (Report UCB/CSD 84/184), Computer Science Division, University of California. Berkeley (California, USA), May 1984.

[51] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, Cambridge (UK), 1986.

[52] S. L. Hurst. *Custom VLSI Microelectronics*. Prentice Hall, Hemel Hempstead (UK), 1992.

[53] W. J. Hutchins and H. L. Somers. *An Introduction to Machine Translation*. Academic Press, London (UK), 1992.

[54] S. M. Kafka. An assembly source level global compacter for digital signal processors. *Proceedings of 1990 International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 1061–1064. IEEE, 1990.

[55] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnik and N. D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, pp. 5–54. Prentice Hall, Englewood Cliffs (New Jersey, USA), 1981.

[56] W. Kent. Assembler-language macroprogramming: a tutorial oriented toward the IBM 360. *ACM Computing Surveys*, vol. 1, no. 4, pp. 183–196. December 1969.

[57] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs (New Jersey, USA), 1978.

[58] B. W. Kernighan and D. M. Ritchie. *The C Programming Language* (second edition). Prentice Hall, Englewood Cliffs (New Jersey, USA), 1988.

[59] D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading (Massachusetts, USA), 1984.

[60] D. E. Knuth and L. Trabb Pardo. The early development of programming languages. In N. Metropolis, J. Howlett, and G.-C. Rota (eds.), *A History of Computing in the Twentieth Century*, pp. 197–273. Academic Press, New York (New York, USA), 1980.

[61] L. Lamport. *LATEX: A Document Preparation System* (second edition). Addison-Wesley, Reading (Massachusetts, USA), 1994.

[62] J. R. Larus. Assemblers, linkers, and the SPIM simulator. In [81], Appendix A.

[63] E. Lassila. ReFlEx—an experimental tool for special-purpose processor code generation. Report B15, Digital Systems Laboratory, Helsinki University of Technology. Espoo (Finland), March 1996.

[64] E. Lassila. A macro expansion approach to embedded processor code generation. *Proceedings of EUROMICRO 96 Conference*, pp. 136–142. IEEE Computer Society Press, Los Alamitos (California, USA), 1996.

[65] P. D. Lawrence and K. Mauch. *Real-time microcomputer system design: an introduction* McGraw-Hill, New York (New York, USA), 1987.

[66] E. A. Lee. Programmable DSP architectures. *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19 (Part I), and vol. 6, no. 1, pp. 4–14 (Part II). October 1988 and January 1989.

[67] P. Lee and M. Leone. Optimizing ML with run-time code generation. *ACM SIGPLAN Notices*, vol. 31, no. 5 (Proceedings of SIGPLAN '96 Conference on Programming Language Design and Implementation), pp. 137–148. May 1996.

[68] J. R. Levine, T. Mason, and D. Brown. *Lex & yacc* (second edition). O'Reilly, Sebastopol (California, USA), 1992.

[69] S. Y.-H. Liao. Code generation and optimization for embedded digital signal processors. Ph. D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Cambridge (Massachusetts, USA), 1996.

[70] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded DSP microprocessors. *Proceedings of 32nd Design Automation Conference*, pp. 599–604. ACM, 1995.

[71] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pp. 235–253. May 1996.

[72] H. Lunell. Code generator writing systems. Ph. D. thesis, Software Systems Research Center, Linköping University. Linköping (Sweden), 1983.

[73] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York (New York, USA), 1974.

[74] P. Marwedel and G. Goossens (eds.). *Code Generation for Embedded Processors*. Kluwer, Boston (Massachusetts, USA), 1995.

[75] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, vol. 3, no. 4, pp. 214–220. April 1960.

[76] J. R. Metzner. A graded bibliography on macro systems and extensible languages. *ACM SIGPLAN Notices*, vol. 14, no. 1, pp. 57–68. January 1979.

[77] P. A. Nelson. A comparison of PASCAL intermediate languages. *ACM SIGPLAN Notices*, vol. 14, no. 8 (Proceedings of ACM SIGPLAN Symposium on Compiler Construction), pp. 208–213. August 1979.

[78] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs (New Jersey, USA), 1989.

[79] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058. December 1972.

[80] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, vol. 28, no. 1, pp. 8–21. January 1985.

[81] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo (California, USA), 1994.

[82] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP design tool requirements for embedded systems: a telecommunications industrial perspective. *Journal of VLSI Signal Processing*, vol. 9, no. 1, pp. 23–47. 1995.

[83] E. Pelegri-Llopart and S. L. Graham. Optimal code generation for expression trees: an application of BURS theory. *Conference Record of 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240. January 1988.

[84] D. R. Perkins and R. L. Sites. Machine-independent Pascal code optimization. *ACM SIGPLAN Notices*, vol. 14, no. 8 (Proceedings of ACM SIGPLAN Symposium on Compiler Construction), pp. 201–207. August 1979.

[85] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Hemel Hempstead (UK), 1987.

[86] J. L. Reuss. Macro implementation of a structured assembly language. *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 284–287. May 1982.

[87] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, vol. 20, no. 1, pp. 160–187. January 1973.

[88] D. Salomon. *Assemblers and Loaders.* Ellis Horwood, Chichester (UK), 1992.

[89] M. Shaw and W. A. Wulf. Towards relaxing assumptions in languages and their implementations. *ACM SIGPLAN Notices*, vol. 15, no. 3, pp. 45–61. March 1980.

[90] E. Skordalakis. Meta-assemblers. *IEEE Micro*, vol. 3, no. 2, pp. 6–16. April 1983.

[91] N. Solntseff and A. Yezerski. A survey of extensible programming languages. *Annual Review in Automatic Programming*, vol. 7, pp. 267–307. Pergamon Press, Oxford (UK), 1974.

[92] T. B. Steel. A first version of UNCOL. *Proceedings of 1961 Western Joint Computer Conference*, pp. 371–378. 1961.

[93] T. B. Steel. UNCOL: the myth and the fact. *Annual Review in Automatic Programming*, vol. 2, pp. 325–344. Pergamon Press, Oxford (UK), 1961.

[94] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, vol. 1, no. 8, pp. 12–18 (Appendix C in no. 9, pp. 9–16). August (and September) 1958.

[95] T. Swan. *Mastering Turbo Assembler* (second edition). Sams Publishing, Indianapolis (Indiana, USA), 1995.

[96] W. L. van der Poel and L. A. Maarssen (eds.). *Machine Oriented Higher Level Languages.* North-Holland, Amsterdam (The Netherlands), 1974.

[97] J. Welch. Structured programming in macro assembly languages. *Software—Practice and Experience*, vol. 8, no. 3, pp. 371–376. May–June 1978.

[98] T. R. Wilcox. Generating machine code for high-level programming languages. Ph. D. thesis, Department of Computer Science, Cornell University. Ithaca (New York, USA), 1971.

[99] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37. December 1994.

[100] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, vol. 30, no. 6 (Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation), pp. 1–12. June 1995.

[101] N. Wirth. PL360, a programming language for the 360 computers. *Journal of the ACM*, vol. 15, no. 1, pp. 37–74. January 1968.

[102] W. A. Wulf. Issues in higher-level machine-oriented languages. In [96], pp. 7–12.

[103] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, vol. 14, no. 7, pp. 41–47. July 1981.

[104] W. Wulf, C. Geschke, D. Wile, and J. Apperson. Reflections on a systems programming language. *ACM SIGPLAN Notices*, vol. 6, no. 9 (Proceedings of a SIGPLAN Symposium on Languages for Systems Implementation), pp. 42–49. October 1971.

[105] W. A. Wulf, D. B. Russel, and A. N. Habermann. BLISS: a language for systems programming. *Communications of the ACM*, vol. 14, no. 12, pp. 780–790. December 1971.