Rule-based Compilation

of data-parallel programs

Rule-based Compilation
of data-parallel programs

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 3 juni 2003 om 10.30 uur
door

Leonard Conrad BREEBAART

doctorandus in de informatica
geboren te Amsterdam.

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. ir. H.J. Sips

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft, promotor
Prof. dr. ir. H.E. Bal	Vrije Universiteit Amsterdam
Prof. dr. W.G. Vree	Technische Universiteit Delft
Prof. dr. H.A.G. Wijshoff	Universiteit Leiden
Dr. A. van Deursen	Technische Universiteit Delft / Centrum voor Wiskunde en Informatica
Dr. E.M.R.M. Paalvast	Cisco Systems International BV
Dr. ir. C. van Reeuwijk	Science & Technology BV
Prof. dr. ir. J.L.G. Dietz	Technische Universiteit Delft, reservelid

ISBN 90-6464-659-7

Author email: leo@lspace.org

To the memory of my grandmothers

Acknowledgements

The road has been rather long — not to mention somewhat winding.

Over the years it has been my good fortune to encounter many people who have given me more of their time, companionship, professional and personal help, and above all: patience than was perhaps warranted by my seeming determination to indefinitely position the deadline for finishing this thesis at "next year".

I would first of all like to thank my promotor, Henk Sips. He not only gave me the scientific support and supervision that a graduate student can expect from his professor, but he also allowed and encouraged me to remain part of the Parallel and Distributed Systems group in Delft long after I had formally left. Thanks to him, I have never been without a desk, a computer, a friendly ear, and the occasional job at the University. Without those I would never have made it this far.

Edwin Paalvast, then of TNO-ITI, was my primary supervisor in the early years. His ideas, his research, and especially his unique brand of enthusiasm form the bedrock on which much of this thesis was built. I cannot help but feel apologetic towards him over the fact that the *Booster* language — so very much a cornerstone of our work in those early days — eventually fell by the wayside...

Kees van Reeuwijk, of Delft University, was my primary supervisor in the later years. I could not have wished for a more thorough discussion partner and sounding board (on *any* subject under the sun), while his excellent software (particularly the Template Manager *Tm* and the compiler backend for *Vnus*) has been instrumental in helping the *Rotan* prototype described in this thesis come to life.

There are three fellow researchers whom I would specifically like to thank for the support they have given me over the years: Peter Doornbosch for his help with the implementation of the earliest *Rotan* prototypes and the definition of the *Rule Language*, Paul Dechering for his work on the definition and semantics of the

Vnus intermediate language, and Frits Kuijlman for his engineering of the *Vnus* runtime system and his help with, especially, the Communication Aggregation rules.

A list that, alas, has far too many names on it to mention separately is that of all the co-workers, group members, and roommates that I have worked, talked, and lunched with over the years. My gratitude goes out to all these colleagues and former colleagues at the TNO Institute for Applied Information Technology (TNO-ITI); the Computational Physics (FI-CP), Pattern Recognition (FI-PH), and Parallel and Distributed Systems (ITS-PDS) groups at Delft University; the Simulators (2.4) and Virtuality (2.5) groups at the TNO Physics and Electronics Laboratory (TNO-FEL); as well as to my brand new band-of-brothers at Science & Technology BV.

A special word of gratitude, finally, to all the members of the *ParTool* project and its various follow-up efforts and spin-offs. The project is finished, and for most of us our ways already parted long ago, but they have been stimulating and exciting times, and I treasure the memories — *we'll always have Garderen.*

Moving towards more personal acknowledgements, I would like to execute a big `blocksend()` of aggregated thanks towards all my family and friends[1] — with a special shout-out to my *paranimfs* Mahin Ramkisor and Melvin Asin — for their help, friendship and patience, and for the fact that they never gave in to the temptation to make fun of my perennial thesis delays and woes. Well — hardly ever.

I am, of course, particularly indebted to my parents and my brother Harold for their monumental, unwavering support and encouragement on all fronts. They have truly always been there for me, and without them none of this would have been even remotely possible.

Delft,
April 2003

Leo Breebaart

[1] You know who you are.

Contents

Chapter 1

Introduction

1.1 Parallel Programming: a Problem?

Parallel computing has been a popular and fruitful computer science research subject for decades, but the long-awaited breakthrough towards mainstream, commercial computing has yet to materialise.

The increased complexity of parallel computing when compared to conventional sequential computing is one factor contributing to this delay. Another is that while parallel *hardware* has over the years shown continuous signs of progress and improvement (in terms of power, price, availability, etc.), parallel *software* has always lagged behind and fallen short of expectations, making it difficult to actually use the hardware in a profitable manner. In a sense, new, sometimes radically different parallel machines are being built before there has been enough time to properly learn how to program the old ones.

Compiler technology is another important factor in parallel computing. We know that programming parallel computers is in general more difficult, error-prone and time-consuming than programming sequential computers. Higher level programming languages explicitly designed for parallelism can be crucial in alleviating this burden, as well as in aiding portability and maintainability. But as these programming languages generally involve abstracting away the complexities of parallel architectures at a lower level, the demands on the compiler technology (and consequently on the compiler programmers) increase. The applications programmer wants to be bothered as little as possible by aspects of the architecture that have no direct relationship to the problem domain he or she is trying to program for. However, a compiler needs to know about and take into account all these aspects if it is expected to generate efficient code out of the higher level specification.

This constant trade-off between the desire for ease-of-use and high levels of

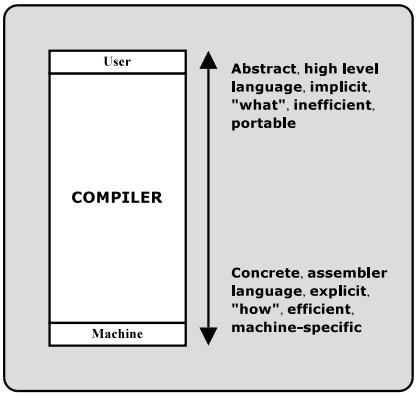


Figure 1.1: *The performance/effort trade-off.*

abstractions on the one hand, and the desire for performance and machine-specific optimisations on the other hand (Figure 1.1) is as ubiquitous (perhaps even more so) in parallel computing as it is in other areas of software engineering.

Compilation software for parallel languages, while not necessarily having to run on the parallel target architecture itself, is also still vulnerable to the demands of constantly evolving hardware, system environments, and programming languages. Maintainability and adaptability become even more important issues than we already know them to be. Just as with programming in general, at least a partial solution might be found in increasing the level of abstraction at which the compiler itself is engineered, and it is with this subject area that the research presented in this thesis is concerned.

1.2 Smarter Compilers: a Solution?

The central research question investigated in this thesis is whether by applying tools and techniques from the fields of rewrite systems and generative programming, we can create a higher level of compiler for parallel architectures, one that is in fact itself (partially) programmable.

Given such a programmable compiler, we can plug into its framework dedicated modules consisting of rewrite rules implementing specific transformations and optimisations (Figure 1.2). This would elevate the concept of the compiler above that of the traditional, monolithic black box written in hundreds of thousands of lines of conventional language code. Compiler writers will be able to react to changes and concerns at both ends of the trade-off spectrum. If a specific algorithm or class of algorithms needs to be written for a parallel architecture, the application of user knowledge about the domain or the algorithm in question can

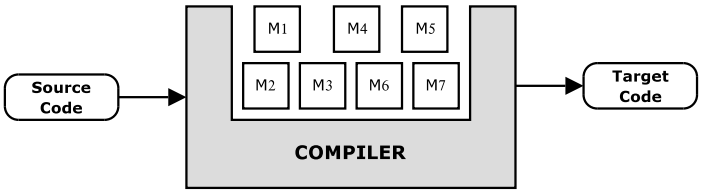


Figure 1.2: *A programmable compiler.*

lead to an expansion of the compiler with new, dedicated optimisation modules. If, on the other hand, existing programs need to be compiled for a new or updated architecture for which the existing compiler does not generate optimal code anymore, these lower-end changes can also be accommodated by adding or changing specific rules. Optimisation and benchmarking become easier and less error-prone, because these efforts can also take place at the higher abstraction level of the transformation rules, thereby evading many of the maintainability and expandability problems that plague more traditional compilers. The compiler-building environment and associated rule transformation language should be powerful enough that such operations can be done in a modular, portable, interactive fashion.

In this thesis we present research into, and the implementation of just such an open, programmable compilation framework, called *Rotan*. This compilation framework includes support for a high-level, rule-based transformation language called the *Rule Language*. Translation and optimisation operations on arbitrary domains can be expressed in this language in a modular fashion, using *rule engines* built out of individual rewrite rules. This allows a level of code-reuse and fine-tuning that is not easily possible to achieve in conventional compilation systems.

While many of the techniques investigated in this thesis have a general applicability to the fields of compiler construction and parallelism, it is only practical to focus our attention on a smaller, representative area. We have chosen the compilation of data-parallel programming languages as our focus. Data parallelism is a programming model in which the distribution of data structures drives the parallelism: the location of the data determines which process is responsible for computations involving that data. Data-parallel languages and compilers are quite successful, and will provide a strong real-world context in which to place (and against which to compare) the research presented in this thesis.

1.3 Thesis Context and Overview

This thesis describes research initially performed in the context of the *ParTool* project, a nationally funded Dutch parallel processing project. The work was continued as part of the activities of the Parallel and Distributed Systems group of the Faculty of Information Technology and Systems (ITS-PDS) at the Delft

University of Technology.

The thesis is outlined as follows:

Chapter 2 (*Data Parallelism*) gives a brief general introduction to parallel programming, followed by a closer look at the data-parallel programming model that will be the focus for the remainder of the thesis. The chapter then introduces and discusses certain key concepts in data parallelism such as SPMD Programming, data distributions, and the designation of computation responsibility.

Chapter 3 (*Compiler Construction Tools*) describes compilation models for sequential and parallel programming, i.e. it introduces the viewpoint of the compiler builder. An overview of existing compilation tools and approaches is given, as well as a review of general-purpose transformation systems, the latter with specific attention paid to their suitability for implementing a compilation system for data-parallel programming systems. A new programmable compiler framework called the *Rotan* system is proposed as a means of obtaining the levels of flexibility, expressive power, and maintainability such a system requires.

Chapter 4 (*The Rule Language*) introduces the *Rule Language*, the rule-based transformation language that forms one of the key components in the *Rotan* framework. It allows the compiler builder to implement translations and optimisations by specifying high-level transformations on the parse tree of a source program. This chapter explains the syntax and gives an informal operational semantics of the *Rule Language* as implemented in the current *Rotan* prototype.

Chapter 5 (*A Rotan Compiler for Vnus*) describes the major test case for the *Rotan* system: an implementation of a semi-automatically parallelising compiler for the *Vnus* language. (*Vnus* is a programming language used as an intermediate format in the compilation process of higher-level (data-)parallel programming languages.) The parallelisation schemes used in this compiler are discussed, and examples of their implementation as rules are given.

Chapter 6 (*Experimental Results*) presents a number of case studies in which the *Vnus* compiler described in Chapter 5 is applied to data-parallel *Vnus* programs. Since a higher abstraction level is generally associated with a decrease in efficiency, this chapter investigates the extent to which this holds true for the target code generated by the *Rotan Vnus* compiler. The performance results of these programs are then compared to those achieved by other compilers.

Chapter 7 (*Evaluation*) steps back from the details of the generated code, and evaluates the experiences with the general *Rotan* system both in terms of its own design criteria as well as in comparison to a different compilation system in use at the Delft University of Technology.

Chapter 8 (*Conclusion*) closes the thesis with a summary and discussion of the presented research topics, concluding with some suggestions for future research.

To aid the reader in following the various *Rule Language* and *Vnus* program fragments given throughout the thesis, Appendix A and Appendix B give the grammars for the *Rule Language* and *Vnus*, respectively. Appendix C contains the *Rotan* domain definition for *Vnus*, expressed in the *Tm* data structure format. Appendix D, finally, lists the largest single rule in the *Rotan Vnus* compiler. This

is intended as an illustration of a truly non-trivial rule, and also conveys an informal idea of the actual 'upperbound' for the rule complexity encountered in the compiler.

Chapter 2

Data Parallelism

2.1 Introduction

In this chapter we will take a short tour of the field of parallel programming in general, and data-parallel programming in particular. We introduce and explain various concepts and terms that will be referred to frequently in this thesis.

We begin with an overview of parallelism in general, before narrowing our focus to the topic of data parallelism. Using a simple vector addition as a running example we illustrate concepts such as *Single Program Multiple Data* (SPMD) programming, and data distribution specifications.

We then go into some of the issues concerning the efficiency of SPMD programs and introduce some of the optimisations, such as owner test absorption and communication aggregation, that will be encountered in more detail in subsequent chapters.

2.2 Parallel Programming

Parallel or distributed programming is a possible solution whenever conventional, sequential programs executed on conventional, single-processor architectures do not yield answers fast enough, or do not allow problems to be modelled with a sufficient level of complexity or detail. Typical examples of this abound in the field of High Performance Computing (HPC), where we find big simulation and modelling applications such as used for e.g. weather-prediction [Rod96; Wol95] and fluid dynamics [Wil02], or massive data processing applications such as World Wide Web search engines [Bri98], or the search for extra-terrestrial life [SET02].

In those and many other cases, it makes sense to try and attack the problem by distributing the computations over different processors, to be executed not sequentially, but in parallel. This "many hands make light work" concept is a

very intuitive and logical one. It exists both on the hardware level, where a set of processors or machines will be able to execute (parts of) programs in parallel; and on the software level, where algorithms can be formulated as concurrently running processes or tasks, at different levels of granularity.

At the hardware level, parallel processing can be implemented within the confines of a single machine, such as seen in low-cost Symmetric Multi-Processor systems [Tum02] or in high-end supercomputers [Cra02; ASC02], but it can also be implemented within a wider system of connected machines such as a Wide Area Network (WAN), or indeed even the Internet. These latter cases are usually described as "distributed computing", but the boundaries between parallel and distributed computing are hard to delineate.

What remains a common element is that in all these cases the most important (often the only) purpose of introducing parallelism is the achievement of *speedup*. People turn to parallel programming because this allows them to create programs that are equivalent in functionality to a sequential version, but which take far less time to run.

In the ideal case, the speedup should be *linear*: if the number of processes concurrently working on the program is multiplied by a factor n, the execution time should decrease by the same factor. In practice, there are a number of pitfalls and problems associated with parallel programming, which make achieving speedup, let alone linear speedup, a non-trivial exercise:

Intrinsic algorithmic limits. Amdahl's well-known law [Amd67] states that speedup will always be limited by the sequential, nonparallelisable parts of an algorithm, and will therefore rarely be linear. While the effects of Amdahl's Law can sometimes be offset to a certain degree by scaling the problem size along with the number of processors, it is indeed worth remembering that many algorithms no not lend themselves well to parallelisation to begin with. Even those programs that are parallelisable need careful consideration — merely adding more processors often will not lead to satisfactory speedup, and may in some cases actually *increase* execution time.[1]

Communication and shared resources. A parallel programming model introduces a number of new issues that are not present in sequential programming. Non-trivial parallel processes need to *communicate* and *synchronise* in order to exchange data with each other, and they will have to share available hardware and software resources (such as buses, memories, and global variables) amongst themselves. This can lead to problems such as deadlock, starvation, and race conditions. Even if these are all avoided on the algorithmic level, the performance of the resulting program can still be affected by issues such as load-balancing and resource contention.

[1] For instance, adding processors can increase inter-processor communication overhead costs to the point where they outweigh the gain in decreased computation costs.

Conceptual programming difficulties. Parallel programming is difficult to do. It is not easy for the human mind to keep track of concurrently executing threads of control. It is also not always obvious what a good parallel algorithm for a sequentially described problem might be.

These considerations make it more difficult for a parallel programs to be correctly written, let alone for them to be efficient and portable. We can speculate that the lack of mainstream acceptability for parallel programming is at least in some part due to the sheer difficulty involved in creating parallel software.

One way of tackling this problem is to develop more advanced compilation techniques in order to help programmers create efficient parallel programs [Per96]. In Chapter 3 we will investigate this in more detail.

2.3 Parallel Programming Models

All parallel computer systems have the presence of multiple processes (or processors, threads, tasks, etc.) in common, but two types of system are often distinguished, based on the memory model:

Shared memory systems. In a shared memory system, the processes all read from and write to the same single memory space. In such systems, data does not need to be distributed or communicated, which makes programming easier. However, memory becomes a shared resource, which means that processors must avoid accessing a memory location at the same time. Shared memory systems also tend to scale badly on the hardware level.

Distributed memory systems. In a distributed memory system, the processes all have access to their own, local memory space. In such systems, some data will need to be distributed, communicated, and kept consistent, which adds to the complexity of the programming. But since memory accesses are local, this approach also avoids much of the resource contention that plagues shared memory systems.

At the hardware level, there are also the Non-Uniform Memory Access time (NUMA) systems, where there are different categories of memories and caches available to a process, each with its own characteristics and access times. These systems increase the scalability of shared memory architectures, and go some way towards bridging the gap between shared and distributed memory.

A property that is often used to categorise parallel programming models is *granularity*:

Coarse-grained parallelism. This model deals with large, self-contained units such as functions or objects. These are scheduled to execute concurrently and will explicitly communicate with each other via e.g. loosely-coupled

message passing. *Task parallelism* [Has96] is a typical example of coarse-grained parallelism.

Fine-grained parallelism. This model typically exploits parallelism that exists within a program at a lower level, for instance at that of statements within a loop, or that of multiple small actions operating on large data sets. *Functional parallelism*, that is: the dataflow type parallelism achieved by analysing a program written in a functional programming language and identifying the subtrees in the resulting dependency graph that can be evaluated in parallel, is an example of fine-grained parallelism.

Instruction-level parallelism. This model exploits the parallelism that exists at the last stage of the compilation trail: the assembler instructions for the program are mapped directly onto the available hardware in a way that optimises resource usage.

Orthogonal to the granularity aspect, parallel programming models can also be described by the *level of explicit parallelism*:

Explicit parallelism. A programming model is explicitly parallel if it is possible (or required) for the programmer to explicitly use scheduling, communication and synchronisation primitives to implement the program's parallel flow of control.

Implicit parallelism. A programming model is implicitly parallel if it allows the programmer to express algorithms in a sequential fashion, with the parallelism inherent in the specification being extracted later on by the compiler.

Programming models are seldom entirely explicit or implicit, and many hybrid forms exist. Implicit parallelism is desirable, because it leads to a programming model that is easier to program in. It is however not generally possible to extract all the inherent parallelism from a program by automatic means at compile time. After all, program flow may be dependent on conditions or values that cannot be not known until runtime. Explicit parallelism tends to lead to more efficient programs, because the programmer has absolute control and can ensure that the available parallelism is fully exploited.

It is no surprise that the performance/effort tradeoff mentioned in Chapter 1 (Figure 1.1) still applies: all parallel programming models try to strike a balance between how much information about parallelism the programmer needs to supply, and how much information the compiler needs to discover by itself.

A particularly successful parallel programming model, and the one this thesis is concerned with, is the *data-parallel* model.

2.4 The Data-parallel Programming Model

We will use the following definition of the data-parallel model of computation:

Definition. *Data-parallel programming is a form of parallel programming in which the programmer specifies the distribution of data over the processors. It is left to the compiler to choose and implement the distribution of the actual computations over the processors.*

Expressed in terms of the properties described in the previous section, data parallelism is a fine-grained, implicit programming model. As the name implies, data parallelism is data-driven: the actual, explicitly parallel computations (and communications, where necessary) are deduced by the compiler from the only aspect that the user is given control over: the distribution of data over the processing units.

The challenge for compiler engineers is not just to create a compiler that derives explicitly parallel programs from any given data distribution specification, but to create one that does so in a manner that leads to an efficient, fast-executing program.

Because of its focus on memory and data structures, data parallelism is well-suited to distributed memory hardware architectures at the machine level. At the algorithmic level, it is applicable to a large number of problem areas, from database applications to array-based numerical applications. It is with this latter area that we are particularly concerned in this thesis: numerical algorithms involving vectors, matrices and other regular, large data structures are often suitable candidates for being implemented as a data-parallel program.

2.4.1 A Data-parallel Example

When adding two vectors of length n, a sequential program might perform the required calculation element by element in a loop:

```
double a[n], b[n], c[n];

for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

With a total of p processors at its disposal, a data-parallel algorithm may instead allocate a part of the vectors to each processor's memory, and then distribute the additions over the processors accordingly. The program fragment executed by each processor would then look like this:

```
double a[n/p], b[n/p], c[n/p];

for (i = 0; i < n/p; i++)
    c[i] = a[i] + b[i];
```

This fragment only shows the core idea of data parallelism. In an actual implementation there are a number of issues to consider, such as how the original

data structures are divided over the processors, how the responsibility for different calculations is divided over the processors, and how actual values and results are communicated between processors.

Also important is the question of how these data-parallel programs are created in the first place. They might be hand-written and customised for each processor, or perhaps be (semi)-automatically generated from an algorithm specification. Such a specification itself might be explicitly or implicitly parallel, or perhaps a combination of both.

It is outside the scope of this thesis to give a complete overview of the field of data parallelism, or to exhaustively consider all the possible approaches that can be chosen to tackle the aforementioned issues. A more comprehensive treatment can be found in e.g. [Hat91].

As will be described in more detail in Chapter 3, this thesis concerns itself with semi-automatic compilation, and focuses on the generation and optimisation of data-parallel programs from an implicitly parallel specification. These programs do not contain any explicit knowledge of different processors or memories, but contain a sequential algorithm that assumes a single global memory space.

For the remainder of this chapter we will focus specifically on those aspects of data parallelism that are relevant to the research described in this thesis.

2.4.2 SPMD Programming

Single Program Multiple Data or SPMD programming is a form of data parallelism in which each processor executes the same, explicitly parallel program, parameterised by processor number, that acts on its own, separate data structures. The processor number parameter is used in various expressions seen throughout the program: array bounds and loop conditions are typical examples.

Because of this parameterisation, individual processors executing an SPMD program can proceed in a loosely coupled fashion, and need not run in strict synchronised lockstep. Each processor executes the same program, but the dependence on the processor number parameter means that the different instances can still be doing quite different things at any given moment in time.

For instance, a pseudo-code SPMD version of the vector addition example given in the previous section might look like the code displayed in Figure 2.1.

In this — highly inefficient — program, we see that the values of the elements in the right-hand side of the central computation, `a[i]` and `b[i]`, are communicated one by one, through paired *send/receive* commands to the processor that 'owns' `c[i]`. That processor will then use the values in the actual computation. In SPMD programming, the *send* and *receive* primitives are typically provided by a communications library linked with the program, such as a Message Passing Interface (MPI) library [Sni96] or a Parallel Virtual Machine (PVM) library [Gei94].

Because the tests involve the parameter p, which will be different for each processor, the actual work done in each iteration of the loop can also be very different. Some processors will be sending data, some will be receiving data,

```
double a[n], b[n], c[n];
extern int p;    // processor number

for (i = 0; i < n; i++)
{
    if (sender(a[i]) == p)
      send(owner(c[i]), a[i]);

    if (owner(c[i]) == p)
      receive(sender(a[i]), tmp_a);

    if (sender(b[i]) == p)
      send(owner(c[i]), b[i]);

    if (owner(c[i]) == p)
      receive(sender(b[i]), tmp_b);

    if (owner(c[i] == p)
      c[i] = tmp_a + tmp_b;
 }
```

Figure 2.1: *Element-wise SPMD version of data-parallel vector addition.*

some may do both or neither, some will compute an addition, some will do no computation at all. Yet every processor executes the same program, which needs only be written, executed, and debugged once. This is a major advantage of SPMD programming.

There are of course still many ways in which the efficiency of our crude example program can be improved.

In terms of memory usage, it is not necessary for each processor to have local arrays of size n. Since each processor is only responsible for storing a subset of the original vectors' data, the local arrays can be shortened to n/p in size (assuming the data structures are distributed evenly over the processors).

In terms of actual communication, the program as given communicates one data element at a time, which is very inefficient. It would be an improvement if the communication could be *aggregated*, with all elements that need to be sent to processor q collected in a buffer that can be sent with a single *blocksend* command.

Also, this program will communicate elements even if these elements are already local to the processor in question. This might for instance be the case if the vectors a, b, and c are all distributed in exactly the same fashion. In such a case, `a[i]` and `b[i]` will already reside on the same processor as `c[i]`, and no communication would actually be necessary at all.

As a last example, it is also inefficient for each processor to loop through the entire index space of n elements. It might be possible to iterate only over the n/p elements the processor in question is responsible for, rather than waste time performing tests on all the other elements. For certain regular forms of data distribution, it is possible to derive a formula for the exact set of elements that p is responsible for, instead of looping over the entire index space and executing expensive run-time tests on ownership. This process is described in detail in [Ree96] and referred to there as *owner-test absorption*.

Implementing these and other improvements is a time-consuming and error-prone task when done manually. However, given certain formalised ways of describing data and computation distribution, they can be derived automatically and incrementally from a starting specification. The development of compiler technology that facilitates this is the primary focus of this thesis.

2.4.3 Data Distributions

In typical data-parallel programming languages such as High Performance Fortran (HPF) [HPF97] or *Spar/Java* [Ree01], the user specifies an algorithm using global address-space data structures, but then annotates the program with hints to the compiler on how the data structures should be distributed over the available memories.

For example, in HPF an $n \times m$ matrix *grid* might be distributed cyclically in the first dimension over a linear processor array by using the special `!HPF$` 'distribute' annotation as follows:

```
double precision, dimension(1:n,1:m) :: grid
!HPF$ distribute grid (cyclic,*)
```

In *Spar/Java*, the same distribution might be specified using the '`on`' pragma and a lambda function:

```
double grid[n,m] <$on = (lambda (i j) P[(cyclic i)])$>;
```

In data-parallel programming it is typically assumed that each available processor has a corresponding local memory. For the remainder of this thesis, we will make the same assumption, and therefore use phrases such as "allocated to a memory" or "allocated to a processor" interchangeably.

Although the specific details will vary from language to language, there are in general five major kinds of data-distribution functions. These are illustrated in Figure 2.2, where each variant is used in turn to distribute a vector v of length $N = 12$ over a processor array p consisting of $P = 3$ nodes.

Block distributed. The data is divided into P blocks of at most $X = \lceil \frac{N}{P} \rceil$ consecutive elements each. Each processor p is then assigned one block. In Figure 2.2, this means that processor p will be assigned elements $p \cdot X, p \cdot X + 1, p \cdot X + 2, \ldots, p \cdot X + (X - 1)$.

Cyclic distributed. The data elements are assigned one by one to the processors in round-robin fashion. In the current example, processor p will end up with vector elements $p, p + 1 \cdot P, p + 2 \cdot P, \ldots, p + (\lceil \frac{N}{P} \rceil - 1) \cdot P$.

Block-cyclic distributed(k). Blocks of k consecutive elements are assigned to the processors in round-robin fashion.

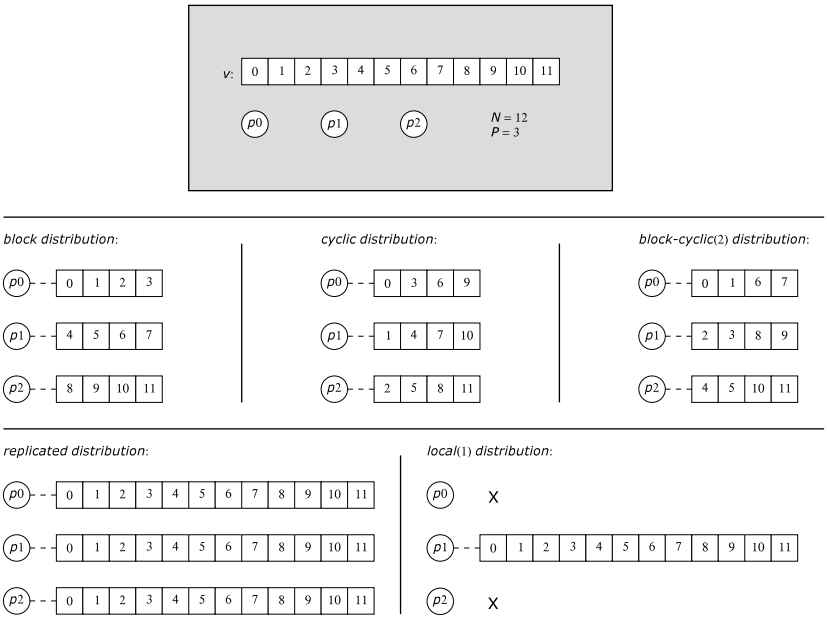


Figure 2.2: *Five forms of data distribution.*

Replicated. Replicated data is available locally to each processor, and need never be fetched from another processor at all. In other words, each processor p is assigned all elements $0, 1, \ldots, N-1$. It is up to the SPMD-generating process to ensure that replicated data remains consistent across processors: if a replicated value changes on one processor, all other copies of the variable in question have to be updated as well.

Local(p). The data in question is local to processor p alone. Local distribution is often used as the default distribution for 'small' data, such as loop counters or accumulators. Again, p is assigned all elements $0, 1, \ldots, N-1$, but there is now no need to synchronise with the other processors (unless the data is explicitly requested, of course): the values are truly local to processor p only.

Both block and cyclic distributions are special cases of block-cyclic distribution. Block distribution equals block-cyclic($\lceil \frac{N}{P} \rceil$), cyclic distribution equals block-cyclic(1).

Allowing explicit block and cyclic distributions is a useful shorthand, because these two variants suffice for many algorithms, and when reasoning about them

they allow useful simplifications of the more complex formulae associated with the generic block-cyclic distribution [Ree96].

Figure 2.2, and our examples so far have all used one-dimensional data structures and a one-dimensional processor array, that is: a set of equivalent processors numbered p_0 through $p_{(P-1)}$. Both data structures and processor arrays can be generalised to multidimensional structures, in which case the situation becomes a bit more complex.

If a two-dimensional matrix is mapped to a two-dimensional processor array, it becomes possible to map non-linear blocks of data elements to a processor, for instance by specifying different distributions in each of the available dimensions.

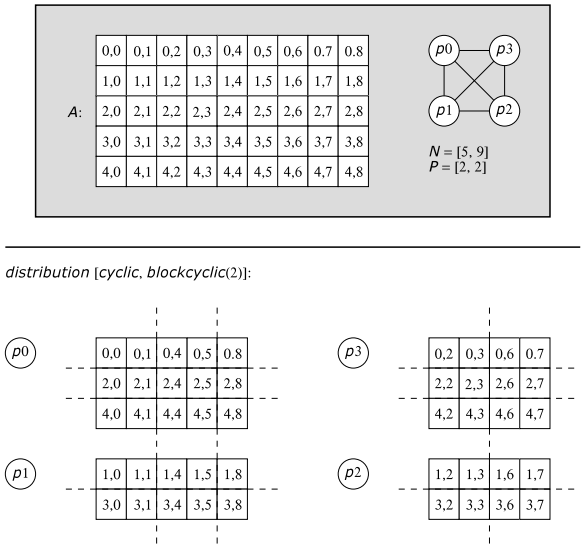


Figure 2.3: *A two-dimensional data distribution for a matrix.*

Figure 2.3 shows a matrix A mapped onto a two-dimensional processor array. The elements are distributed cyclically in the first dimension, but block-cyclic(2) in the second dimension. Every processor ends up with a smaller sub-matrix of elements.[2]

For the remainder of this thesis, we will consider a one-dimensional processor array only. This is sufficient for many different algorithms and applications. It is

[2]In this example the matrix dimensions each have an odd number of elements whereas the processor dimensions are even. This illustrates a load-balancing problem that can be inherent in all non-block distributions: processor p_0 ends up with almost twice as many elements as p_2, and will presumably have to do nearly twice as much work. These effects can be avoided by choosing a more suitable distribution.

true that the use of multi-dimensional processor arrays open up more possibilities for parallelism specification or program optimisation, but when it comes to the compiler technology needed to implement such higher-dimensional specifications, not that much more (other than the extra complexity) is added to the concepts and rewrite rules that will already cover for the one-dimensional case.

We will still continue to consider multidimensional data structures, however. With a one-dimensional processor array, such a structure can only be distributed along one of its dimensions. Thus, it is possible to for instance distribute a matrix column-wise or row-wise, by specifying a distribution for the first or for the second dimension. The non-distributed dimensions are said to be *collapsed*: the elements in these dimensions will be allocated to the same processor as was specified along the corresponding distributed dimension. Figure 2.4 illustrates this.

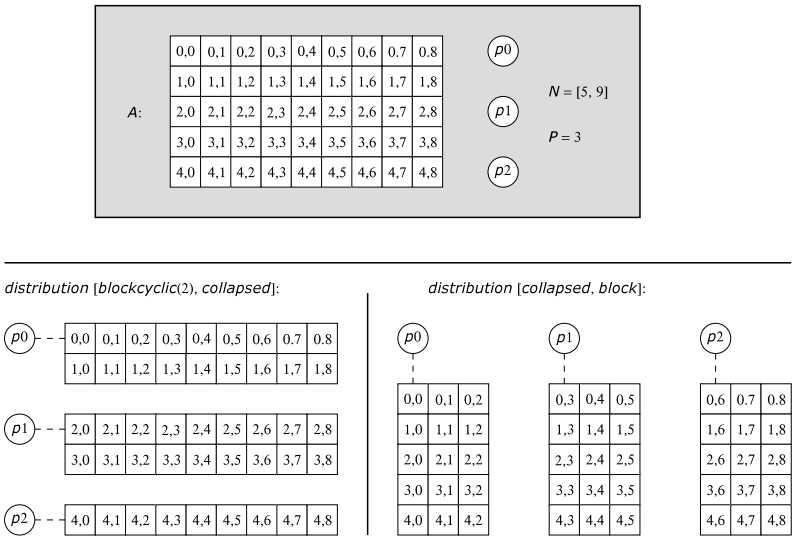


Figure 2.4: *Collapsed data elements.*

2.4.4 Computation Responsibility

An SPMD program must also somehow specify which of the available processors is responsible for which computation.

One possible solution is to allow the programmer to specify explicit computation distributions, analogous to the data distributions. Data structures, procedures, as well as specific (parts of) statements can be annotated with directives or hints concerning the responsibility of computation.

A less explicit, and far more ubiquitous approach is to use the so-called *owner-computes* rule, which equates computation responsibility with data responsibility. This is the approach that has been followed in the examples given so far. The processor that owns the data element on the left-hand side of a specific statement is in charge of performing the computation that determines this new value. If the computation uses data elements that do not reside on that processor, those data elements must be retrieved from the processor where they do reside.

There are other approaches still. Computation responsibility could for instance be assigned to the processor that already owns the majority of the data elements occurring in the computation, in order to minimise communication. Or a compiler may choose to have some computations performed by multiple processors, also with the same goal. And as with data distribution, it can also be left up to the programmer to specify an exact location for certain computations by adding an annotation or pragma to the original computation code. Such approaches are outside the scope of this thesis, however, and for the remainder of this document we will assume that computation responsibility follows from data ownership: the owner computes.

2.4.5 Conclusion

The data-parallel programming model has been one of the more successful approaches to parallel computing to emerge from the last few decades of scientific research and commercial development. It is flexible, powerful, and offers a level of abstraction that is a workable compromise between high- and low-level concerns: abstract enough for programmers to prefer using it over e.g. explicitly parallel variants, low-level enough for compilers to be able to generate acceptably efficient code.

Nevertheless, it remains true that this programming model presents many interesting and new problems to the compiler builder. Writing compilers for parallel languages in general, and for data-parallel languages specifically, is far from a 'solved' problem, successful attempts such as the various High Performance Fortran compilers [Per96] notwithstanding. It is therefore a fruitful area in which to try new approaches to compilation.

In the next chapter, we will focus on the parallel compilation model rather than the parallel programming model, and investigate techniques and approaches that can help ease the task of creating compilers for (data-)parallel programming languages.

Chapter 3

Compiler Construction Tools

Compilers are large and complex software systems. Over the years, many tools for generating compiler components from higher level specifications have been developed. These tools can aid compiler writers in getting a grip on the complexity, and thus decrease the development effort involved in writing compilers.

In this chapter we investigate existing types and categories of tools. We then focus on transformation tools specifically, and will formulate some desirable properties for such tools to have. We then compare and contrast a number of existing systems in terms of these characteristics. In the final section of the chapter, we introduce our own entry in the field: the *Rotan* system.

3.1 Compiling Sequential Programming Languages

Figure 3.1 shows the traditional compilation path on sequential architectures for transforming a higher level source program into machine-executable target code.

The *parsing* phase is comprised of *lexical analysis*, in which the text of the source program is converted into syntactical tokens, and the actual *parsing*, in which an abstract syntax tree is created from the token stream, and then converted to a semantically equivalent representation in some intermediate format.

In the *analysis* phase, the parse tree is traversed and additional semantic information about the program is collected. Typical examples of analyses are the marking of the places in a program where variables are defined and used (def/use analysis), or the determining of the types of expressions (type inference).

In the *transformation* phase a wide spectrum of further analyses and optimisations are applied to the program tree. Taken together, these make up the actual transformation of the source program into a different domain.

The *synthesis* phase maps the transformed program tree to a target output stream. It is in this phase that final machine-dependent transformations and

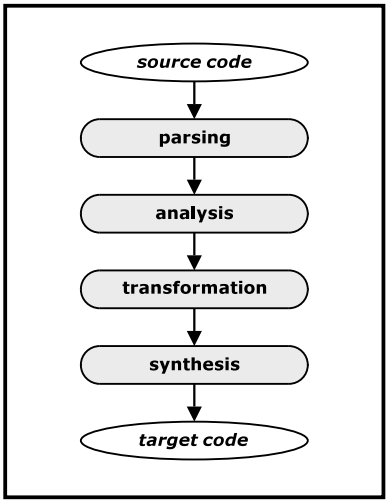


Figure 3.1: *The compilation path.*

optimisations are performed, such as register and memory allocation.

In general, the parsing and analysis phases are performed by software components collectively referred to as the compiler *frontend*, and the synthesis phase by a component called the compiler *backend*.

The four compilation phases are not rigorously defined, and the boundaries between them are often blurred. Also, there are some forms of analysis and transformation which do not occur just once, but which are repeated throughout the compilation process. Examples of this are performing analysis during parsing phase by decorating the parse tree with inherited or synthesised attribute values, and the use of a constant-folder to simplify expression trees at various times during the program tree's life span.

Even though many of the algorithms and patterns used in compiler design are well-known and long-established, the creation and especially the maintenance and expansion of compilation systems remains a difficult and time-consuming task. Given this relatively static nature of the compiler state of the art, one might have expected that by now writing compilers would be a solved problem, in which standard libraries and code reuse can extensively be used to easily piece together a compiler from predefined modules.

In reality, this is not the case. There are always crucial differences involved that make what works well for compiling one specific source language to one specific architecture not suitable for compiling a different language to a different architecture. In particular, different languages often need different internal formats in order to fully exploit the advantages of the languages during compilation, and this too, makes code reuse more difficult.

Nevertheless, there are patterns and frameworks in each of the compilation phases that have emerged over time as stable and useful in many cases, and over the years compiler builders have created and used tools and technologies that allow specifications at even higher levels of abstraction than offered by high-level programming languages.

Often these tools are not so much libraries of predefined manipulations, but rather higher-level code-generating programming tools, where that which is unique to each compiler can be expressed by the compiler writer in a language of a kind, while that which is the same will be generated or taken care of by a run-time system.[1].

Such tools are very common in parsing, frequent in analysis and synthesis, but rarer in the more complex and less clearly defined transformation phase.

Parsing

For the parsing phase, *compiler construction tools* have become commonplace. Generators for lexical analysis, such as *lex* [Les75] and *flex* [Pax95], allow the specification of a language's tokens as a list of regular expressions from which a scanner is then generated. Parser generators such as *yacc* [Joh75], *bison* [Don90], *CUP* [Hud96], *ANTLR* [Par95], and *JavaCC* [Web02], allow the specification of a grammar as a list of production rules, from which a parser for the language defined by that grammar is then automatically generated. The so-called scannerless parser SGLR [Bra02] does away with the need for a separate lexical phase altogether.

In all these cases, deriving parsing tools automatically from higher-level specifications offers the compiler writer advantages similar to those which any higher-level programming language offers: increased ease of use and more flexibility, expandability, maintainability, and portability. The focus can be on specifying the actual language to be scanned and parsed (the 'what'), rather than on specifying the logic that performs the scanning and parsing (the 'how'). That logic is generated from established templates that can be considered constant factors that no longer need to be explicitly created (and debugged, and maintained) every time they are used.

Analysis

There are a few automated tools that can be used in constructing systems for program analyses such as dataflow and type inference systems. One such tool is the Berkeley Analysis Engine BANE [Aik98], another is the commercial static-data analyser PAG [Mar98].

In general, analysis is so closely entwined with the transformation phase that it is common to see an analysis pass approached as a 'read-only, side-effect free'

[1] Many of these meta-programming techniques can be seen as examples of an approach that has recently become known as *generative programming* [Cza00].

type of transformation, expressed in whatever formalism or tool is used during that phase.

Transformation

The transformation phase is less well-defined than the parsing or synthesis phases are. It is the component where in typical compilation systems most of the complexity and non-standard algorithms are located.

Nevertheless, this phase also lends itself to a specification at a higher level of abstraction, The most frequently encountered mechanism being that of *rewrite systems* that allow one to specify a transformation in terms of pattern matching rules with associated actions. These systems take care (to varying degrees) of generating the code for traversing the program tree (the so-called *treewalkers*) and applying the transformations. We will discuss transformation systems, including our own system *Rotan*, in more detail in Section 3.3 of this chapter.

Of great importance to the transformation phase is the intermediate format used to represent the program. If the format is not powerful enough to express the semantic information that is present in the source program, and if analysis results cannot be stored and retrieved in an accessible manner, the amount of optimisation that the transformation engines will be able to perform will be limited.

Synthesis

In the synthesis phase, too, there has been an increase in the level of abstraction over the years. This can be seen from languages such as *twig* [Tji86] and *beg* [Emm89], where high-level descriptions of target machines' instruction sets allow the automated generation of backends for different target architectures, or from systems such as *Machine SUIF* [Hal96], that allow a modular specification of instruction optimisations, independent of compiler environment or even compilation target.

In line with this trend for increased abstraction levels, there has been also been a move, especially in experimental compilers, towards more wide-spread use of source-to-source translations. Here the target language is no longer assembly code or an intermediate format, but rather a higher programming language such as C or C++. In this way the already available, existing compilers for these languages can become the backend generators, freeing the compiler writer for the source language from having to consider machine-dependencies or assembly-level optimisations (at the expense of some efficiency loss). Source-to-source translations are often used in those cases where the source language itself is at a still higher abstraction level than the target language (e.g. fifth generation languages, or parallel programming languages such as *Spar/Java*).

A third abstraction approach often encountered in synthesis is that of the intermediate framework, in which the main compiler compiles to an abstract architecture that is itself platform-independent. For different target platforms there

will be custom implementations of a backend for the intermediate framework. Examples of such frameworks are *p-code* [Pem82] and the *Java Virtual Machine* (JVM) [Lin99] (both explicitly designed for a single front end language), and the *Architectural Neutral Distribution Format* (ANDF) [Dia94] and *.NET* [Mic02] (both of which also abstract the front-end by being designed for many different source languages).

3.2 Compiling Parallel Programming Languages

When compiling parallel programming languages for parallel architectures, the level of sophistication required from the compiler increases.

Parallel programming languages are often high-level, with a large amount of implicit semantics. Examples of this are shared, global address spaces that need to be converted to distributed local memories in the target code; and parallel loop constructs such as *foreach* or *forall*, which have complex semantics that cannot always be mapped one-to-one to the available lower-level primitives in the target language [Dec97a; Dec97b; Dec96].

Parallel machines offer more degrees of freedom than sequential architectures do. In typical cases there will be multiple processors and multiple memories; and these can be connected in a large variety of different ways. As we saw in Chapter 2, the generated code needs to take into account issues such as communication and synchronisation.

Another factor is that on a parallel architecture it is rarely sufficient to write code that 'merely' executes correctly — it also has to run *fast*. As a result, the optimisations performed in the transformation phase of the compilation play an even more important role in the parallel case than they do in the sequential case, and are often more complex as well (or at the very least will cause a considerable increase in the *number* of simpler transformations).

Because of the focus on optimisation and target code efficiency, the parallel compilation trail needs to support many opportunities for tweaking and performance tuning by the programmer. This begins at the source level by the presence of language constructs that allow the user to give hints or instructions to the compiler as to how certain constructs can be interpreted or optimised. This can for instance be done in the form of *pragmas* or *annotations* as used in languages such as *Booster* [Paa91; Bre91; Bre95], HPF [HPF97] or *Spar/Java* [Ree01]. The intermediate format used by the transformation system also needs to be powerful and expressive enough that these hints can be properly stored, interpreted, and accessed.

Furthermore, it is desirable for the system to provide sufficient hooks to allow the compiler writer to create and experiment with new optimisations, and to be able to tweak and rearrange existing ones.

A high-level transformation system based on rewrite rules is, with these points in mind, a good candidate for a solution. In such a system, optimisations can be

programmed individually, can be reordered and reused, and the compiler writer can focus on the optimisations themselves rather than have to be concerned with issues such as low-level, explicit data structure traversal and modification.

Apart from the expressiveness of the system, another requirement is that the compiler writer should be offered a programming environment that allows iterative development of rules.

A final requirement is that it should be possible for the compiler writer to 'escape the system', and have the option to extend parts of the optimisation process by applying external code that is not expressed in the transformation system itself.

Taking all this together, we can list a set of desirable properties a transformation system should have for it to be considered powerful enough to be used in a compiler for a (data-)parallel language. It should be noted that these properties are not specific to compiling parallel languages as such — in practice one can expect them to serve any complex compilation goal equally well.

1. The system has to be rule-based. Transformations are specified as "*this* becomes *that*", preferably using a specification that is not tightly coupled to a specific host language. This allows the system to be used in many different contexts.

2. The actual mechanics and implementation of applying the transformations (i.e. traversal and manipulation of the code tree) should, to as large an extent as possible, be generated or handled by a run-time system; the programmer should not have to explicitly write code for it.

3. Rules should be modular and easy to reuse at different places in the compilation trail, similar to functions in a conventional programming language.

4. The system should be interactive, so that hands-on experimentation with rules is possible, and transformations can be defined (and debugged) iteratively, leading to shorter development times.

5. The intermediate format used by the system should to a large extent be definable or at the very least accessible by the compiler writer. It should be generic, yet powerful enough that it can support the many different semantics that will be found in the categories of languages that it will be used to encode. Many compilation systems require the compiler writer to use one specific intermediate format, which is then sometimes entirely internal to the system and not accessible or even visible to the compiler writer.

6. The transformation language should abstract from the intermediate format that its programs act upon in the sense that different formats should be supported. This leads to the concept of a parameterisable, typed, rule-based language, in which the *framework* of the language remains the same, but where different domains can be plugged into that framework to yield

a new language for every new intermediate format, as long as that format complies with some general rules (i.e. expressible as some combination of tuples, lists, and expressions).

7. It should be possible for the compiler writer to 'escape' the high level rule language, and have access to the intermediate tree for other kinds of processing. This can involve delegating the tree to be processed by completely different tools or components, or simply specifying part (or all) of a transformation rule in a lower-level language than the rule language itself.

In the next section, we will describe some existing systems, and investigate their suitability for implementing a data-parallel compiler in, in the light of these requirements.

3.3 Transformation Systems

There are many different language/tree transformation systems, all with their own focus and positioning along various axes of comparison (programmability, performance, application domain, etc.).

In general, higher levels of abstraction provide more ease-of-programming, lower levels of abstraction provide more expressiveness. Many interesting characteristics of transformation systems (maintainability, efficiency, etc.) can be directly correlated to the system's positioning on this primary axis.

All transformation systems start from the assumption that coding transformations by hand is cumbersome and error-prone, and that at least *some* forms of additional abstraction and/or predefined (run-time) support are necessary to reduce the complexity of the task.

The transformation systems generally make use of a specific host language (e.g. C, C++, Java), which the system then either extends with additional primitives and keywords, or translates to, or both (many systems are implemented as preprocessors).

Most systems provide some form of support for all the stages in the compilation process, from parsing (lexical scanning, parse tree and abstract syntax tree generation), to data structures (common formats for interchange between components or with the outside world), to the transformations themselves (rules and strategies), to output (code generation, or just pretty printing).

The early 90s saw the appearance of many monolithic systems, in which the aforementioned forms of support were all locked into a single encompassing custom framework that tried to be all things to all people. Recent developments see these large systems more and more being replaced (or augmented) by component-based systems that allow the user varying degrees of freedom when it comes to choosing different components, or interfacing with software not directly provided by or expressed in the system [Ber98].

It is not the intention of this chapter to give a complete review of transformation languages and systems. The field is very widespread, touching upon many application areas from computer algebra and theorem proving to language specifications and code generation. A recent, global overview of rewrite-based systems can be found in [Hee02], while [Vis01b] provides an interesting taxonomy of program transformations.[2]

In the following pages, we will take a look at some of the better-known or more interesting transformation systems, with a focus on their suitability for translating and optimising data-parallel languages.

TXL

TXL [Cor02] is a mostly functional, rule-based transformation language. The system implements its own lexer and parser for context-free grammars. Externally defined data types cannot be incorporated, everything has to be specified using the *TXL* syntax. The data structure format is internal, and not explicitly specified. There is no way to access the data structure itself.

Transformations can be programmed in a rich language of pattern-matching replacement rules. Subrules and rule-sequencing are both possible, as is the linking in and application of user-defined (sub)rules not written in *TXL*. However, these rules will have to implement their own parsing — *TXL* import and exports only strings. Both global variables and parameters are available to pass data between rules. Many predefined rules exist.

As mentioned, *TXL* only outputs strings. Pretty-printing is completely under the user's control, and can be specified along with the parser rules.

TXL's support for polymorphism is not very elegant. *TXL* is still very much a monolith — despite the possibility of external rules, it is clear that integration with other components is not *TXL*'s strong suit.

TXL could very well be used for creating a data-parallel compiler, but the lack of control over the intermediate format is a serious limitation.

App

The *App* system [Nel00] consists of a C++ compiler with embedded *App* constructs for defining algebraic data types and matches on these types.

The defined algebraic types are mapped directly to various C++ constructs, making use of templates, the Standard Template Library (STL) and Run-Time Type Information (RTTI), amongst others. Integration with user code is obviously possible: user-defined types can be directly used in the algebraic type specification, and there are provisions for incorporating fragments of C++ code into the data types.

[2] A good Internet resource for information on language transformation systems in general is the *Program Transformation Wiki* website [Vis02], whereas for specific compilation purposes the *Catalog of Compiler Construction Tools* website [GMD02] is worthwhile.

The 'match' constructs allows the programmer to abstract from having to manually implement case-based dispatch and tree traversal administration. *App* generates the dispatching framework (complete with dynamic casting etc.). Patterns can be positional or non-positional. Since the actual traversal function is under the user's control and merely gets 'expanded' by *App*, arbitrary traversals can be implemented.

App is a system that is very close to actual C++ code — no power is lost. Its strength is mainly in the area of type definition, the matching and traversal are there, but it appears fairly rudimentary in that much of the administration and treewalking still needs to be implemented manually.

App could be well-suited to write a data-parallel compiler in, but the transformation logic can easily get entangled and lost in the C++ code. In this respect, App is *too* close to C++.

Tm

Tm [Ree92; Ree03b] is a generic template manager and macro expansion language, capable of generating arbitrary code for many different host languages in many different contexts.

The *Timber* compilation system is an example of a dedicated transformation system (in this case: an implementation of a compiler for the data-parallel language *Spar/Java*) that uses *Tm* extensively in its transformation phase.

In *Timber*, most transformation rules are specified in *Tm* templates, from which the C++ code for corresponding treewalkers is then automatically generated.

From an abstraction viewpoint, *Tm* code specifications are close to the target template language. This is good from the point of view of allowing the compiler writer access to the program tree at all times, but does mean that the a lot of domain knowledge is necessary, and that rules are not always as easy to write and maintain as would be the case if the template constructions were available as higher-level abstractions in a language of their own.

We will encounter *Tm* in more detail later in this chapter, when we discuss how *Tm* is used in the implementation of our own *Rotan* system.

Prop

Prop [Leu97] implements its own lexer and parser for context free grammars. Algebraic datatypes can be specified in a functional manner — trees, DAGs and graphs are all possible, and there are constructors for tuples and records. User-defined datatypes can be incorporated by encapsulation (giving them the proper access functions etc.).

Like *App*, *Prop* is a C++ preprocessor. The algebraic types are mapped to C++ classes. This means that after parsing, the syntax tree (and a large amount of generated support) is fully available to the programmer as a tree of instances of these classes.

Also like App, *Prop* defines a switch-like 'match' construct for implementing (conditional) type-based dispatches of user-specified code. It provides more sugaring and more power, e.g. repetition and cost minimisation are both supported, as are rule variables, and vector and list forms. Orthogonal to the pattern matching, simple inference classes are supported, along with rewrite rules that act on them. Constructs for garbage collection and object persistence are available. In addition to the simple pattern matching, concrete rewrite rules (both replacing and applicative) are supported.

The greatest disadvantage of *Prop*'s wide range of supported functionality is that it is a complex, kitchen-sink type of language. It adds over a *hundred* new keywords to C++. *Prop* is no longer actively being developed.

The ASF+SDF Meta-environment

ASF+SDF [Bra01] works on syntax definition formalisms expressed in SDF (Syntax Definition Format), i.e. a context-free grammars.

Its internal representations are called *ATerms* — the annotated trees that form the interchange format between all of the ASF+SDF tools.

Transformations are specified as equational rules, which the main system applies until no further reductions are possible. There is no facility for specifying deterministic rule strategies, but since the introduction of the Toolbus architecture it has become possible for other components that act on ATerms (such as *Stratego*) to be used to that effect.

Escaping from ASF+SDF rules to external code, or even linking to it, is not possible. The first versions of ASF+SDF were very monolithic systems. In combination with Toolbus, it has become more modular, but its focus is still mostly as a system for interactive prototyping.

The needs of data parallelisation and optimisation are not particularly well-served by the non-deterministic ASF rules, which can also has negative implications for the performance of the system. The more recent ASF+SDF components such as *JJForester* and *Stratego* are more promising in this respect.

JJForester

JJForester [Kui01] acts on a syntax definition formalism in the SDF format (i.e. a context-free grammar).

Its intermediate format is a Java class hierarchy, corresponding to the datatypes defined in the specification. Internally, ASF+SDF's ATerms are used.

JJForester does not have direct support for transformations (i.e. no 'match' or 'rule' constructs — it does not extend the Java language at all), but it generates tree traversal methods of which the 'action' part for each type encountered can be filled out as desired by the programmer (by inheriting from, and then refining methods of, generated base classes). The companion *JJTraveler* framework for implementing *visitor combinators* makes it possible for the programmer to have

complete control over the tree traversal and application sequence of the various transformations.

The generated Java code, which together with user-supplied code forms the program that will act on the actual input and generate the actual output. There is no special output support.

JJForester is more a component than a real transformation system. It helps hide the complexity of programming tree traversal functions by hand, and allows the programmer the full power of Java-level programming, but actual transformations need to be manually programmed. *JJForester* truly becomes useful when applied in conjunction with other tools, such as the *JJTraveler* framework, or the various *XT* tools.

Stratego

Stratego [Vis01a] acts on a text string that specifies (and will be converted to) an ATerm.

Stratego modifies the input term by applying rewrite rules. Rules are basically simple one-step global applicative rules of the kind used in most term rewriting systems, but *Stratego* offers a complete meta-language of user-definable rule strategies, that allow fine-grained control over rule-application flow.

Stratego's main power is that it brings determinism to the world of term rewrite systems, with a correspondingly powerful matching mechanism. However, escaping to lower-level code is not possible, and externally defined rules cannot be invoked.

XT

XT [Jon01] is a diverse collection of tools. Typically, a syntax definition in SDF will be the starting point.

ATerms are the common tree exchange format between all the individual *XT* components.

XT is primarily a bundling of a number of existing tools, many of which can also be found in the ASF+SDF environment. *XT* allows a more command-line oriented, pipelined approach towards generating a compiler. The primary transformation components in *XT* are *JJForester* and *Stratego*.

XT is far more extendible than ASF+SDF, and allows easy interfacing with components written in other languages. It is a very coarse-grained framework — it is the individual components that are of the most interest here.

XT by itself does not do any transformation, but is used as the glue that connects components together and makes them form a compilation system.

mtom

mtom [Mor01] acts on user-defined terms in any formalism (e.g. ATerms or XML).

Its intermediate format is any user-defined data structures in the goal language for which the appropriate interface methods (access, equality, etc.) have been specified.

mtom is a true preprocessor. Its constructs are specified directly in the host language program, as yacc-like constructs. User-defined data types can be wrapped in an API that allows mtom to access them (after access and equality functions, etc. are defined). mtom only provides support for rules with pattern matching, reasoning that the entire right hand side is best left expressed in the host language. User-defined evaluation strategies are supported.

Like *Prop*, this is another rather low-level abstraction layer on top of the host language (typically C++) that takes care of generating some of the more tedious code associated with pattern matching and rewrite rules. There is no real support for rule strategies, though. The emphasis in mtom is specifically on supporting the user's data structures. It is therefore a good tool for integration with existing code. mtom programs defined to work on ATerms could easily be a component in *XT*, for instance.

The mtom rule construct lacks some of the power that would be necessary for comfortably specifying data-parallel optimisations and transformations.

JastAdd

The *JastAdd* system [Hed01] works on a parse tree that an external parser (in this case JavaCC/JJTree) must generate for an abstract context-free grammar specification in *JastAdd*'s custom formalism,

Its intermediate format is a custom Java class hierarchy, corresponding to the datatypes defined in the specification.

Like *JJForester*, *JastAdd* generates treewalkers that can be used by user-defined visitors. It employs code weaving to add user-defined methods defined in separate modules directly into the parse tree classes. It also has support for modules that allow classic, declarative attribute grammar specifications to be woven into the generated code.

JastAdd is another low-level system that does not support transformations directly, but generates a lot of code that supports the application of generic code to the tree in various helpful manners. The combination of visitors and class weaving is interesting.

JastAdd has no concept of rewriting rules.

Maude

Maude [Cla99] works on input strings that are parsed by the system itself. The usual algebraic types are possible, including object oriented types that use inheritance.

The intermediate format is unspecified, and not directly accessible in any way.

Transformations are specified in terms of equational and rewriting logic. Algebraic types are specified and rules are listed. The system applies these rules until no further simplifications are possible. It is not possible to install strategies to direct the rule-application in a deterministic fashion (as alternative to the usual non-deterministic rule applications). No escape to a lower-level language is possible.

Maude is not a system that is very well-tailored for data-parallel program transformations — it is clearly powerful, but needs everything expressed in its own formalism, which is too generic and high-level.

The abstract syntax tree is directly manipulable, but there is little control over the rules, and there is no way to escape to C++ and act on the tree directly.

Conclusion

A problem with many of the systems described above is that they are either generic transformation systems, not explicitly tailored towards being used for compilation of data-parallel programming languages, or that they are low-level systems that lie so close to the host language that not sufficient abstraction and run-time support are offered, and the ease-of-use and flexibility suffers.

This is particularly reflected in the fact that many of the intermediate formats used are not well suited to the intermediate format necessary for adequately describing data-parallel programs in.

3.4 The *Rotan* System

In this section we describe the *Rotan* system: a compiler construction tool specifically targeted at the transformation and optimisation phases of the compilation process [Bre92]. It was specifically built to possess all the desirable traits listed in Section 3.2:

1. The system is rule-based, with individual rules expressed in a high-level language called the *Rule Language* (described in detail in Chapter 4).

2. The *Rotan* run-time system interprets the rule and applies it to a given parse tree. The compiler writer can choose different modes of application and tree traversal, but needs to specify these actions explicitly.

3. Rules are identified by name, and can be recursively ordered and aggregated into higher-level units called *drivers* and *engines*.

4. The *Rotan* system contains a command-line environment with built-in support for working with and debugging rules.

5. *Rotan* works on a generic higher-level intermediate format known as a *domain*. Domains are generic enough that they can be used to express a wide

spectrum of intermediate formats, yet specific enough for the system to be able to act on them efficiently. The underlying implementation of the intermediate format is up to the compiler writer.

6. The *Rule Language* is parameterised: each specified domain will automatically lead to a customised version of the *Rule Language* in which new attributes and types become available that correspond to nodes in the domain.

7. The *Rule Language* has a mechanism for incorporating C++ code into the rule itself. This can be used to 'escape' from the higher level and perform dedicated tasks, or call external libraries, etc. The full intermediate tree is available to these functions (although it is of course now up to the programmer to ensure that e.g. the parse tree remains valid — with greater power comes greater responsibility).

3.4.1 Creating a Rule-based Compiler

Figure 3.2 illustrates the way in which a custom compiler for a specific domain or language is created using *Rotan*.

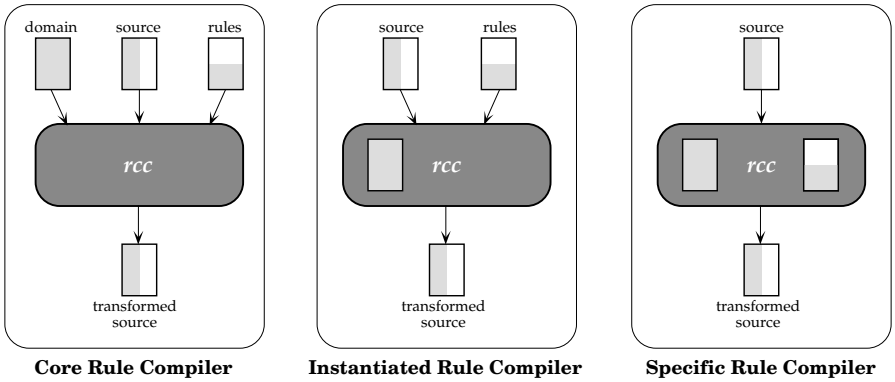


Figure 3.2: *The Rotan compiler framework.*

Step 1. The term *rcc* stands for *Rotan Command-line Compiler*. This is the outer-most *Rotan* framework. The *rcc* is a command-line environment and run-time system with an associated plug-in architecture that generically supports loading, applying and debugging rules.

Step 2. Once a domain is defined (by creating a so-called *domain file*, and by either writing or generating a corresponding parser and pretty-printer), the result is a specific *instance* of the *Rotan* system, parameterised to this domain. The command-line environment will now be able to read and write

programs written in the domain language, and rules can now be written using the types and data structures defined by the domain.

Step 3. Now that we have an instantiated *Rotan* compiler, we can add a collection of actual rules that implement a specific transformation, which can be anything from a single optimisation to a complete translation. This final result is referred to as a Specific *Rotan* Compiler, and yields a program that can actually be used as a command-line compiler that will process any given source program.

3.4.2 Domains and *Tm*

In the *Rotan* system, the format used to specify a domain is that of the *Tm* data structure specifications [Ree00a]. These specifications define a collection of data types which can have inheritance and member-of relationships to each other. Generic collection structures are supported in the form of lists and tuples.

For example, the specification of a top-level *Vnus* program is as follows:

```
Vnusprog == NObject +
                (pragmas:[Pragma],
                 declarations:[Declaration],
                 statements:Block);
```

Vnusprog is a type that inherits all data members from the type *NObject*, and in addition has three data members of its own: a list of elements of type *Pragma*, a list of elements of type *Declaration*, and a member of type *Block*.

Each of the types mentioned can either be an external type (defined as a C++ class somewhere), or a type itself defined in the domain file. For instance:

```
Block == NObject +
            (scope:String,
             statements:[LabeledStatement]);
```

This defines the type *Block* as inheriting from *NObject* as well, with a scope and a statement list as data members. The full domain data structure file for the *Vnus* language is given in Appendix C.

After a data structure definition file has been created, the next step is to write one or more so-called *Tm template files*.

A template file can contains arbitrary text that will be copied verbatim to an output file when the template is run through the *Tm* executable. However, in the template file the programmer can also specify *template language* commands for the *Tm* macro preprocessor. In the template language all lines starting with a dot (‘`.`’) are commands. Such lines are *not* copied to the output, but interpreted by *Tm* instead. In both commands and normal template text, expressions starting with a dollar sign (‘`$`’) are expanded. These expressions can denote arithmetic expressions (e.g. `$[42 + 666]`), variable references (e.g. `$v` or `$(count)`) and function invocations (e.g. `${sort foo bar baz}`).

Most importantly, the dollar variables and template commands can refer to the types and names defined in the data structure file. For example, if we had a domain file for *Vnus* that consisted of the two type definitions mentioned earlier, *Vnusprog* and *Block*, the following template:

```
.foreach t ${typelist}
#include "${tolower $t}.h"
.endforeach
```

would generate the following output:

```
#include "block.h"
#include "vnusprog.h"
```

(The *Tm* function `${typelist}` expands to the list of all types defined in the domain file.)

In this fashion we have written special-purpose templates for *Rotan* that expand our type definitions into the source code for an associated class hierarchy in C++.

This class hierarchy is then used to implement the domain's abstract syntax trees and various operations upon them, such as tree traversal and rule application.

The following excerpt from the template file for the *Vnus* class hierarchy illustrates how templates are used in the generation of a *Rotan* compiler:

```
.. class.ht -- template file for generating C++ header files.
..
.. include file containing constants and variable definitions:
..
.insert dom.t
..
.. iterate over the types defined in the domain file:
..
.foreach t ${typelist}
.if ${member $t $(generated)}

.. create a file for each type:
..
.redirect ${tolower $t}.g.h
..
#ifndef ${toupper $t_g_h}
#define ${toupper $t_g_h}

class ${capitalize $t} : public ${capitalize ${inherits $t}}
{
  protected:

.. Define a corresponding C++ data member for each type data member.
..
.foreach m ${telmlist $t}
.if ${eq ${ttypeclass $t $m} "single"}
    ${capitalize ${ttypename $t $m}} *p${capitalize $m};
.else
    List *p${capitalize $m};
.endif
.endforeach
```

```
  public:

.. Create a constructor method.
..
    ${capitalize $t} (
.set sep " "
.foreach m ${telmlist $t}
.. data members of this type
.if ${eq ${ttypeclass $t $m} "single"}
        $(sep) ${capitalize ${ttypename $t $m}} *in${capitalize $m} = NULL
.else
        $(sep) List *in${capitalize $m} = NULL
.endif
.set sep ,
.endforeach
    );

.. The following functions are be present for every type/class:
..
    int NrDescendants() const;
    Object *Descend(const int) const;
    Object *Descend(const int, Object *);

    static Object *New();
    bool isEqual(const Object& o) const;
    Class *IsA() const;
    Object *Clone() const;
    void PrintSource(Text&) const;
};

#endif

.endredirect
.endif
.endforeach
```

The *Tm* function `${inherits $t}` expands to the direct superclass of the type stored in variable `$t`; the function `${tolower $t}` converts the string stored in `$t` to lowercase characters, and so on.

For the *Vnus* type *Block* the above template generates the following C++ class header in a file *block.g.h*:

```
#ifndef BLOCK_G_H
#define BLOCK_G_H

class Block : public NObject
{
  protected:

    String *pScope;
    List *pStatements;

  public:

    Block (
          String *inScope = NULL
        , List *inStatements = NULL
    );

    int NrDescendants() const;
    Object *Descend(const int) const;
    Object *Descend(const int, Object *);
```

```
    static Object *New();
    bool isEqual(const Object& o) const;
    Class *IsA() const;
    Object *Clone() const;
    void PrintSource(Text&) const;
};

#endif
```

The constructor method *Block()* can be used to create a node of the abstract syntax tree during parsing. The various *Descend()* methods are used to traverse the tree when searching for pattern matches, and the *Clone()* method is e.g. used when applying a transformation.

The actual template code used in *Rotan* is more extensive, and also takes care of e.g. generating `#include` directives, forward class declarations, and accessor functions for all data members. The C++ implementations for these methods are also generated using *Tm* templates, with the exception of the pretty-print *PrintSource()* method which contains functionality that cannot be deduced from the type definition file. Its contents must therefore be explicitly specified by the user.

In addition to these class headers and sources, we also use template files to generate an instantiation of the *Rule Language* for the *Vnus* domain. This includes a specification (in *lex/yacc* format) of a parser that accepts the types and identifiers of the domain as keywords, and generates program trees for rules. This makes it possible to write a rule that will search a *VnusProg* syntax tree for occurrences of a *Block* node in the subtree rooted at its *declarations* field (in order to e.g. perform function-to-procedure conversions).

These components taken together make up the Instantiated Rule Compiler. The final step is to create the actual rules that specify the transformations we are interested in for this domain. The next chapter describes the *Rule Language* used in the *Rotan* system for this purpose.

Chapter 4

The *Rule Language*

In Chapter 3 we have described the *Rotan* framework. In this chapter we will describe the *Rule Language*, the language in which the rules for an actual *Rotan* compiler instantiation are described.

The *Rule Language* allows the specification of transformations on tree-like data structures (such as for instance parse trees). These transformations take the form of sets of rewrite rules.

4.1 Rules and Domains

The *Rule Language* is a hierarchical language. Its smallest unit of functionality is the rule. Rules can be grouped and ordered into drivers; drivers grouped and ordered into engines. In terms of conventional imperative programming languages, engines can be compared to modules, drivers to functions, and rules to statements.

A rule *acts* on a data structure called a *domain tree*. Each rule tries to *match* a *pattern* to a substructure of the domain tree. If a matching structure is found, we say that the entire rule *matches*, and will then *fire*. This means that the matched substructure will be altered or *processed* according to a recipe also described in the rule. If a match cannot be found, the rule is said to *fail*. The following pseudo-code describes a generic rule:

```
rule rulename ;
begin
match pattern  // What to search for
→
action pattern  // What to replace it with
end.
```

A domain tree is a set of connected, user-defined *nodes*. The manner in which the nodes are connected comprises the *structure* of the domain tree. There are two

basic, domain-independent structure constructs in the *Rule Language* for building a domain tree: the *list* and the *expression*.

A list is a finite sequence of *elements*. A list element can itself be another list (with other lists as elements, and so on), an expression, or a node. Heterogeneous lists are allowed.

An expression is a construct consisting of an *operator* plus a number of *operands*.

Operands can themselves be other expressions (with other expressions as operands, and so on), or a lists, or nodes. Heterogeneous expressions (consisting of e.g. a binary operator that acts on one list operand and one expression operand) are allowed. Operators cannot be lists or expressions, but have to be nodes derived from the intrinsic *Rule Language* node type **`Operator`**.

Nodes form the leaves of the domain tree, and are defined beforehand by the creator of the domain. As we have seen in the previous chapter, each domain is defined by its own set of node types. For example, a domain for a numerical calculator might contain *Integer* and *Real* nodes; a domain for a programming language might contain *Statement* and *Procedure* nodes. In *Rule Language* patterns a node is characterised by an identifier that denotes the *type* of the node, as specified in the domain definition file. Rules are written in a domain-instantiated version of the *Rule Language*, where the node types form a set of fixed entities that can only be used, not changed.

Node types are under the control the domain programmer. In the implementation of the *Rule Language*, types are mapped to C++ types (user-defined or intrinsic). Complex types, such as tuples and classes, are allowed. In the *Rule Language* this creates the possibility of nodes that contain references to other nodes, lists, or expressions (see Section 4.3.2).

Expressions, lists, and the set of nodes belonging to a specific domain are all generic, system-defined building blocks as far as the *Rule Language* is concerned. It is up to the process that generates the domain tree to take semantics into account and ensure that the domain tree describes a valid string in the language the domain represents. The *Rule Language* can be used to program generic transformations that are not by themselves guaranteed to lead to a valid result tree, but the *Rotan* system does implement type-checking (using information about the relationship between types (such as inheritance) learned from the domain definition) to prohibit transformations that would break the type validity of the domain tree. Moreover, it is not allowed to transform a list into an expression, or a node of type A into type B, if B is not derived from A. In this sense, the approach of the *Rule Language* is similar to that of e.g. parser-generators such as *yacc*.

4.2 Basic Patterns

A rule specifies a certain *match pattern* for the system to look for in the given domain tree. Since we want to be able to search for any kind of entity that may

occur in the domain tree, a match pattern can be either a list, an expression, or a single node type, as the following (simplified) grammar rules show:

pattern:
    list
    expr
    node

list:
    < $pattern_{list}$ **>**

expr:
    operator **(** *pattern* **)**

node:
    domain-defined-type-identifier

operator:
    domain-defined-type-identifier

4.2.1 Lists

List patterns contain one or more space-separated patterns between `< >` brackets. Assuming a domain that contains the node types A, B, X, Y and Z, the following pattern denotes a three-element list, with the second element another list:

```
< A < X Y Z > B >
```

4.2.2 Expressions

Expressions are written in a functional notation: an operator, followed by one or more comma-separated patterns (the operands) between ( ) brackets. Valid expression patterns are (assuming an appropriate domain that defines the operators and the nodes):

```
IfOp(A, B, C)               // Ternary expression
MergeOp(<A B C>, <D E F>)   // Binary expr. with list operands
Add(Mult(B,C), Minus(D))    // Nested expression
```

There are infix shortcuts for unary and binary expressions:

```
ReverseOp <A B C>
<A, B, C> Add <D, E, F>
(B Mult C) Add (Minus D)  // Parentheses force evaluation order
```

Parentheses `( )` can be used for grouping purposes in both lists and in expressions. In list contexts such a grouping is referred to as a *sublist*.

4.2.3 Nodes and Operators

Nodes and Operators are both terminals of the *Rule Language*: types that are extracted from the domain definition file. When in the examples in this chapter node type identifiers such as `A`, `B`, or `Statement` are used, the presence of a domain in which these nodes are defined is assumed — none of these identifiers are part of the core *Rule Language*. The same applies to operators. Identifiers such as `Mult`, `Add` and `Minus` are always examples of user-defined operator types.

For operators it should be kept in mind that any precedences that may be present in the domain itself (e.g. a multiplication operator takes precedence over an addition operator), are not reflected in the *Rule Language*, where all operators have equal precedence and are matched left-to-right as they appear in the match pattern. *Rule Language* operators are left-associative.

For example, a parser for a 'calculator' domain might transform the string:

```
A * B + C
```

into a domain tree T of the form:

$$+(*(A, B), C)$$

When writing rules for such a domain, however, the rule pattern:

```
A MultOp B AddOp C
```

will be parsed as:

$$\text{MultOp}(A, \text{AddOp}(B, C))$$

and will therefore not match the domain tree T given above.

An appropriate evaluation order can always be forced using parentheses. A pattern that *does* match the domain tree T is:

```
(A Mult B) Plus C
```

4.2.4 Wildcards

When a rule is applied, the matching mechanism will traverse the domain tree until it fails or finds a part of the domain that conforms to the match pattern. The root of this subtree is then replaced with a new subtree created according to the specification in the action pattern of the rule.

The traversal of the matching mechanism starts at the root of the tree, and is left-to-right, depth-first.

For example:

```
rule R ;
begin
(X Mult Y) Plus (Z Mult Y)
→
(X Plus Z) Mult Y
end.
```

Rule R implements a parse tree simplification on (arithmetic) expressions, using the distributive property. An occurrence of the match pattern is sought in the domain tree and replaced by the subtree specified in the action pattern.

In support for more generic, node-independent structural rules, the *Rule Language* supports the following *wildcards* constructs:

- For lists:
  - `...` (ellipsis). The ellipsis token is used to signify zero or more arbitrary elements within a list. If we have the domain tree:

    ```
    <<A B C> <D E F> <G H I> F>
    ```

    then the pattern:

    ```
    <A B C>
    ```

    will only match `<A B C>`,

    ```
    <D ...>
    ```

    will only match `<D E F>`, and:

    ```
    <...  F>
    ```

    will match both `<D E F>` and the main list itself.
  - **`*`**/**`+`** (repeat constructs).
    The repeat constructs '**`*`**' and '**`+`**' are *Rule Language* postfix operators. They can only be applied within the context of a list, to the list elements. The '**`*`**' matches zero or more occurrences of its argument, the '**`+`**' matches one or more occurrences of its argument.

- For expressions:
  - **`any`**, **`leaf`**, **`noleaf`**.
    The keyword **`any`** used in an expression pattern (it can only be used as an operand, not as an operator) matches an arbitrary expression (i.e. of arbitrary depth and contents) in that position. The keyword **`leaf`** only matches patterns consisting of a single node (but still of arbitrary type) and the keyword **`noleaf`** complements this and matches anything that *is not* a single node. If we have the following domain tree:

    ```
    Mult(Mult(A, B), Mult(D, E))
    ```

Then the pattern:

```
Mult(A, B)
```

will only match `Mult(A, B)`, and nothing else. The pattern:

```
Mult(leaf, leaf)
```

will match both `Mult(A, B)`, and `Mult(D, E)`; the pattern:

```
Mult(any, any)
```

will match `Mult(A, B)`, `Mult(D, E)`, *and* also `Mult(Mult(A, B), Mult(D, E))`; and finally the pattern:

```
Mult(noleaf, noleaf)
```

will match only `Mult(Mult(A, B), Mult(D, E))`.

- For nodes:
  - Wildcard functionality for nodes is implicitly present in the domain hierarchy as specified in the domain definition file.
  - A domain designer can specify in the configuration file that `A`, `B`, and `C` are node types derived from the node `X`. Patterns containing a reference to node `X` will match actual occurrences of either `A` or `B` or `C`. In the context of the *Vnus* computer language domain, an example would be the node type *ControlStatement*. This type can be used in rules to match nodes of type *IfStatement, WhileStatement* as well as *RepeatStatement*.
  - The keyword **`object`** signifies a system-defined node wildcard that will match every node type defined by the domain.
- For operators:
  - Operators (like normal Nodes) can be derived from other operators in the domain. Rules containing a reference to an operator *ArithmeticOp* could match a number of nodes such as *MultOp*, *MinusOp*, etc., depending on how the domain is defined.
  - The keyword **`operator`** is an intrinsic wildcard that will match every type derived from type *Operator*.
  - Ellipses can be used in the operand list of expressions to signify an arbitrary number of operands, e.g. '`Op(A, ...)`', or even '`Op(...)`'. The other repeat wildcards (i.e. `*` and `+`) are not allowed in operator argument lists.

All the wildcard nodes can only be used within match patterns, and are not allowed inside action patterns, where they would be meaningless.

4.3 Rule Variables

The *Rule Language* allows the rule-writer to attach names to arbitrary parts of the matched structure. These names, or *rule variables*, can then later on be referred and assigned to in the action pattern. This makes it possible to specify rules that have more complex results than just replacing the entire matched sub-tree with something else.

4.3.1 Initialising Rule Variables

Naming parts of the matched structure is done by assigning a name to the corresponding construct in the match pattern. It is customary, but not required, to use all-caps rule variable names. The syntax is:

namedPattern:
    pattern `.` *rulevarName*

ruleVarName:
    identifier

This *rule variable binding* has a high precedence in the *Rule Language*. If a rule variable name needs to be attached to e.g. an entire expression, grouping parentheses should be used:

```
A Mult B.N
```

will cause N to refer to the matched node `B`, whereas:

```
(A Mult B).N
```

will cause N to refer to the entire matched binary expression.

Rule variables must be unique identifiers that do not clash with any of the domain's keywords or with the *Rule Language*'s own reserved words — there is no separate namespace for rule variables.

The following rule describes a more generic version of the expression simplification rule given earlier:

```
rule R ;
begin
(X Mult any.L1) Plus (X Mult any.L2)
→
X Mult ($L1 Plus $L2)
end.
```

The match pattern is still replaced by the action pattern, but that action pattern now uses the rule variables $L1$ and $L2$, thereby reusing these parts of the

matched structure. Rule variables are referenced by preceding the rule variable name with a dollar sign.

Use of explicit rule variables is the *only* means by which parts of the matched structure can be reused in the action pattern. In all other situations, the rule execution mechanism will newly create the nodes that appear in the action pattern. (For example: the operator nodes *Mult* and *Plus* as well as the node `X` in the example above).

A rule variable can be used more than once in an action pattern, but there will still be only one physical copy of the data structure it refers to. If this structure is changed by a subsequently applied rule, that change will be seen everywhere.

4.3.2 Attributes

So far, nodes have been described as the atomic 'leaves' of the *Rule Language*. However, when specifying a domain, the domain designer also has the possibility to define, for each type of node, an arbitrary number of *attributes* associated with that node. Each attribute has a name and a type. At the C++ implementation level, these attributes correspond to named fields in a tuple or class type.

For example, consider a node called *Cardinality*. This node type might have attributes *Name* (of type *String)* and *Upperbound* (of type *Expression*). The match pattern:

```
Cardinality
```

will match all occurrences of *Cardinality* in the domain tree, but one of these Cardinalities might have a Name attribute with the value i and an Upperbound of 15, while another match might have the Name j and an Upperbound of 30.[1]

The rule system is configured to extract (from the domain configuration file) attribute-specifying keywords for use within the *Rule Language*, just as it does for the nodes themselves. Attributes are incorporated into a match pattern by using the *where-clause* construct.

A match pattern where-clause consists of a number of *where constructs*, concatenated by boolean operators **`and`** and **`or`** or their equivalent 'operator'-notations: **`&&`** and **`||`** (these are all actual *Rule Language* operators, not user-defined ones), and enclosed in *where delimiters*: two square **`[ ]`** brackets. In a where-clause two basic actions can be described:

Retrieval of an attribute. An attribute's value can be bound to a rule variable. This is the second way of initialising a rule variable (the first was associating a rule variable with a pattern using '**`.`**').

[1] *Rule Language* attributes are perhaps unfortunately named in that they are only partially similar to the attributes known from the classic attribute grammars. In our context, attributes are more like 'named children' that are fully part of the actual parse tree, than like constructs separate from the actual parse tree — although *Rule Language* attributes certainly can be used in such a fashion.

```
node.N [ $R = SomeAttr and $S = OtherAttr ]
```

If this pattern matches, then `$N`, `$R` and `$S` all refer to parts of the matched structure: `$N` to the actual node; `$R` and `$S` to two of its attributes. Attribute binding is only allowed in match pattern where-clauses. In the action pattern rule variables can only be references or replaced.

Testing of an attribute. An attribute's value can also be *compared* to the value of a rule variable. This is only allowed in the match pattern where-clauses. The comparison results in a boolean value. If **`false`**, this will cause the rule to fail immediately. If **`true`**, the matching process for that rule will continue. The syntax is that of C/C++ comparisons. Example:

```
FunctionDef [ $N = Name ] // rulevar binding

    ......

FunctionCall [ Identifier == $N ] // rulevar test
```

This rule will only succeed if the pattern matches *and* if the *Identifier* attribute of the matched FunctionCall node is identical to the name of the matched FunctionDef node.

Testing can be done between all combinations of rule variables and attributes. Next to **`==`**, the other standard relational operators (**`!=`**, **`<`**, **`<=`**, **`>=`**, **`>`**) are available.

The actual *meaning* of these operators is defined in the C++ code associated with the entity. Thus, the example above is only legal if the type of the attribute Identifier has a valid *operator==* defined for it.

The *Rule Language* system has built-in knowledge of two intrinsic node types that occur particularly often in many domains. These are the types *String* and *Int*. It is possible to compare rule variables or attributes directly to literal constants of these types:

```
Cardinality [ Name == "i" && Upperbound == 4 ]
```

Rule variable testing makes it possible to create a final, completely generic version of the simplification rule based on the distributive property for expressions:

```
rule R ;
begin
((any.A Mult any.L1) Plus (any.B Mult any.L2)) [ $A == $B ]
→
$A Mult ($L1 Plus $L2)
end.
```

4.4 Subpatterns

So far we have discussed basic patterns made up of expressions, lists, and typed nodes, possibly with attributes. Attributes to a node can themselves be of a node type, but can also be structural, e.g. expressions or lists.

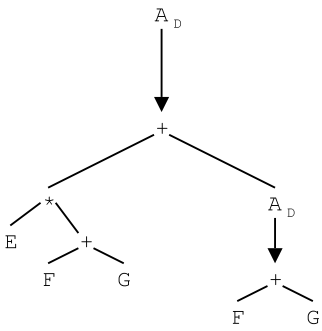
The *Rule Language* supports the specification of constructs that can descend 'into' matched attributes, lists and expressions, and search for the occurrences of subpatterns at arbitrary levels.

These subpattern specifications take place inside *where-clauses*, and use the **`matches`** and **`contains`** keywords:

The 'matches' Keyword

```
node [ Attr matches pattern ]
```

A **`matches`** clause is a construct that allows the rule writer to specify a subpattern for matching against an attribute's exact structure. For example:



This tree consists of a node A, which has an attribute D of type *Expression* (there may be other children as well, but they are not shown in this picture). This expression contains another instance of A with a different D-attribute. The match pattern:

```
Add(F, G)
```

would match both occurrences of '$F+G$' in this domain tree. But the pattern

```
A [ D matches Add(F, G)]
```

only matches the second, bottom-most node A.

The 'contains' Keyword

```
node [ Attr contains pattern ]

node [ contains pattern ]
```

The **contains** clause is used for a less restricted form of matching.

A **contains** clause associated with an attribute Attr to a node *node* matches if *node* is found, and if the Attr attribute of that node contains the specified pattern somewhere inside of it. In the previous example, for instance, the pattern:

```
A [ D contains Add(F, G) ]
```

will match three times: upper A and left $F + G$, upper A and right $F + G$, and lower A and right $F + G$.

Any **contains** clauses associated only with a node will work in a similar way, but will try to find the **contains** pattern in *each* of the attributes of matched node A, as well as in node A itself.

Where-clauses (as described in Section 4.3.2) can be associated with any kind of pattern, not just with nodes. The where-clause can be seen as a postfix operator that applies to an entire pattern: where-clauses can be attached to lists or expressions. In these cases the semantics of the where-clause apply to all the elements of the list, or all the operands of the expression (i.e. the entire expression or list is considered in the search).

There is one restriction: lists and expressions do not have attributes, so a **matches** clause has no meaning here, and neither does a **contains** involving explicit attributes. Thus we get the following:

```
<A, B, C, D> [ $R = XA ]                // Meaningless, error
<A, B, C, D> [ contains (C Add X).R ]   // Ok
(X Mult Y) [ XA contains C ]            // Meaningless, error
```

The where-clause, like the rule variable naming operator '**.**' binds strongly. For example:

```
X Mult Y [ contains (A Add B) ]
```

is a valid pattern, which will search for '(A Add B)' in Y only, not in X. Using parenthesis to change the pattern to '(X Mult Y)' would cause the search to take place in the entire binary expression.

4.4.1 Sublists

Where-clauses can be attached to nodes, lists, and expressions. If a subexpression is enclosed in parentheses, a rule variable or a where-clause can also be assigned to this subexpression.

Similarly, a name or a where-clause can be attached to a sub-list:

```
< X (Y*) [ contains A ] Z >

    ......

< X (Y*).RV Z >
```

The placement of where-clauses relative to the parentheses is important and can affect the meaning of a pattern:

```
< X (Y [ contains T ])* Z >

< X (Y*) [ contains T ] Z >
```

The first pattern matches zero or more successive Y nodes, each of which contains a T. The second pattern matches zero or more successive Y nodes, *at least one* of which contains a T.

4.4.2 Concatenated Where-clause Terms

Where-clause terms in match patterns can be used to construct boolean expressions using the boolean *Rule Language* operators **and**, **or** and **not**.

If a where-clause expression evaluates to **true**, the pattern matches, if it evaluates to **false** it does not, and the rule fails. A where-clause consisting of a rule variable binding returns **true** by default, rule variable/attribute testing returns the result of the test, and **contains**/**matches** constructs return true or false based on whether the pattern specified in them matches or not. The boolean operators can be used with all these where-clause terms.

For example:

```
Cardinality [ $LWB = Lowerbound and $UPB = Upperbound ]
```

always matches (and allows the rule variables $LWB and $UPB to be used elsewhere in the match pattern), but:

```
Cardinality [ $LWB = Lowerbound and $UPB = Upperbound
              and $LWB == $UPB ]
```

only matches if the values of *LWB* and *UPB* are equal.

4.5 Action Patterns

Whereas the match pattern specifies the pattern to look for, the action pattern specifies what should happen afterwards. A successful match supplies two items that can be used in the subsequent action part:

- A collection of rule-variables that provide the 'hooks' into the matched structure. By referring to rule variables in the action pattern we can restructure or modify these substructures, and create a new structure out of them.

- The root of the matched structure, which is always bound to the internal rule variable **$top**.

Next to these items derived directly from the match pattern, Nodes, Expressions and Lists are the other constructs that can be used in an action pattern.

For example, the action pattern:

```
($X AddOp <A $B (C MultOp D)>)
```

leads to the construction of a structure consisting of a new expression, whose left operand consists of a piece of the matched domain structure, whose operator is a newly created *AddOp* node, and whose right hand side consists of a newly created list with three elements: a new node A, another piece of the matched structure (via rule variable $B), and a new expression containing the new nodes C and D. This entire new structure then gets assigned to **$top**, so that it replaces the matched structure, which subsequently becomes unreachable.

4.5.1 Where-clauses in Action Patterns

Where-clauses and attributes are also available for the creation of more complex data structures. In an action pattern, retrieving or testing an attribute (as found in a match pattern where-clause) makes no sense. Instead, where-clauses are used to assign new structures to attributes:

Assignment to an attribute. An attribute's value can be replaced by part of the original tree, by means of previously matched rule variables. The syntax is the counterpart of the retrieval syntax, and uses the standard way of referring to a rule variable's value. For example:

```
$CARD [ Name = $ID and Lowerbound = $EXPR ]
```

Assuming that $N had previously matched a node of type *Cardinality*, this action pattern will cause the subtree starting at that node to be reused. After the rule has finished, the attributes *Name* and *Lowerbound* will have new values, but other attributes that were not explicitly mentioned in the pattern (e.g. *Upperbound* or *Stride*) will have retained their previous values.

Some other example of action patterns:

```
($X Add <A B [ BAttr = $R and BAttr2 = N ]>)
```

This pattern assigns a value to two of the new B's attributes: a part of the matched tree to *BAttr*, and a freshly created node N to *BAttr2*.

```
($X Add <A B [ BAttr = (X Mult C) ]>)
```

This is similar to the previous example, only now the attribute *BAttr* (which has to be of type *Expression*) is assigned an entire, freshly created expression.

The boolean operators **or** and **not** have no meaning in action where-clauses. Since there is no success or failure attached to an action pattern where-clause, we only allow the **and** operator, which simply has the semantics of sequencing.

4.5.2 Rule Variable Usage

Every where-clause in a match pattern introduces a new context to which the rule-variables defined in that where-clause are held to belong.

After definition, rule variables may be used anywhere in the match or action pattern from that point on, with the exception of rule variables defined in **or** and **not** clauses, which may not always have a value outside the immediate context in which they were defined. Similarly, rule variables cannot be used as left-hand side values in the action pattern unless the where-clause that holds the assignment *directly* belongs to the same context in which the rule variable was originally defined. For example:

```
<
  AssignStatement.A [ contains Functioncall.F ]
  IfStatement.I [ contains Functioncall.G ]
>
→
$I [ $F = $G ] // illegal assignment
```

The assignment to `$F` is not allowed because `$F` was defined within the context of `$A`, not of `$I`.

A second application of the context rule is illustrated by the following example:

```
AssignStatement.A [ contains Cardinality.C ]
→
AssignStatement [ $C = ...]
```

Here, the use of AssignStatement in the action pattern causes a new node to be created. Since C was initialised within the context of the specific AssignStatement node bound to A, the attempt at assignment within the *new* AssignStatement is meaningless, (the new AssignStatement need not even contain a Cardinality), and is therefore not allowed.

4.6 Embedded Code

The *Rule Language* as presented so far is not capable of describing every possible transformation rule one would like to program. In particular, there are two places

where a programmer might want to do more than is possible within the *Rule Language* itself.

The conditions for failure or success in the match part of a rule might sometimes involve more than the simple equality testing supplied by the *Rule Language*. One would really like a more generic *condition function*, a function that can take the various attributes and rule variables as arguments, and which will return failure or success depending upon arbitrarily complex tests performed on these arguments.

For example:

```
IfStatement.I [ $COND = Cond ] and $COND evaluates to 'true'
→
IfStatement.I [ Cond = Boolean [ Value = "true" ] ]
```

In this rule, the intention is to simplify an IfStatement by reducing its condition to **`true`**, if possible. This is only legitimate if the expression tree referenced by `$COND` actually evaluates to true. Depending on the complexity of the expression this may not always be possible to determine using *Rule Language* rules, or it may simply be more efficient to have this calculated by more powerful symbolic manipulation code.

In this example, the condition function would be the possibility to have `$COND` examined by an external module of code.

Likewise, sometimes one will need to perform an action on a matched structure that is more complex than just building a new piece of domain tree out of rule variables and new nodes, and a *construction function* might be necessary.

In order to achieve maximal expression power for rules a mechanism has been adapted for allowing the rule programmer to construct arbitrary condition and construction functions for rules: the use of *embedded code*. Embedded code is straight C++ code inserted into the rule, with full access to all rule variables and to the domain tree.

Embedded code is specified by text delimited by curly { } brackets. Such *Embedded Code Blocks* (or *ECB*s for short), like rule variable initialisations and where-clauses, can be suffixed to any arbitrary pattern.

In the most extreme case we can envision a rule that consists *entirely* of C++ code, plus a reference to the rule variable **`$top`** — the effect will be similar to writing a direct C++ function that takes the root of the domain tree as an argument.

In more sensible cases, the rule programmer will use the *Rule Language* as much as possible and only 'escape' to embedded code if a condition or construction function cannot be expressed within the language itself.

The *Rule Language* does not parse or execute embedded code itself. Therefore, a rule containing embedded code must always be translated by a C++ compiler, and the resulting object code linked back into the transformation system before the rule can be executed. Before the embedded code is written out to C++, it

does get preprocessed by a simple textual filter. This allows us to use rule variable syntax in embedded code (e.g. references to `$R`) — the filters will replace them with a valid C++ identifier.

The existence of this preprocessing filter allows the addition of a few non-standard bells and whistles to the embedded code. In particular:

- In ECBs that occur within the match pattern, one can use the special short-circuit keywords **`fire`** and **`fail`**. The keyword **`fire`** will cause the rule to match immediately, and is very useful in order to avoid cascading if-statements in case of complex tests. Similarly, **`fail`** causes the rule to fail without any further testing. If the flow of control reaches the end of a condition ECB without encountering an explicit **`fire`** or **`fail`**, the default action is to **`fire`** the rule. An example match pattern with ECB:

```
Cardinality [ $UPB = Upperbound and $LWB = Lowerbound ]
{
  if (::relative_prime($LWB, $UPB) == true) fail;
  // if not:  fire is implied
}
```

- Embedded code in match patterns can, but should never be written to result in actual changes to the parse tree. Such side effects could cause the matching phase to result in a changed tree *even though the rule itself fails*, and thus complete subvert the higher-level semantics of the *Rule Language*.

It is also possible to create and instantiate new rule variables from within an ECB, and then use these rule variables *outside* of the ECB, i.e. in the normal *Rule Language* context. If this is done, then the rule programmer *must* declare to the *Rule Language* system the *type* of this new rule variable.

Type declarations take the form of a single statement outside the rule body, as follows:

```
rule rulename ;
type type1 type2 type3 ...
begin
match pattern
→
action pattern
end.
```

In addition to declaring the types, the rule variables themselves must also be explicitly declared inside the ECB (this time for the C++ compiler's sake). Lists and Expressions never have to be mentioned in the **`type`** declaration — they are always known to the system. Any other additional types that get used in ECBs, even if they have nothing to do with nodes or attributes (e.g. using a method of a class), must also be declared in the **`type`** declaration.

4.7 Multiple Matches

We have described the process of finding and subsequently transforming one specific 'match', but in several of the given examples we have already encountered patterns that could match *several different* parts of the domain tree. This section concerns itself with the question of how choices between different matches are made, and it explains special *Rule Language* constructs for influencing the matching process.

If we take the domain tree:

(A Mult C) Add (B Mult C)

and the following rule:

```
(any ArithOp any).E
→
$E
```

then it is obvious that this rule will fire, since there are three possible matching trees:

1. `$E == A Mult C`
2. `$E == B Mult C`
3. `$E == (A Mult C) Add (B Mult C)`

Whether just one, or a sequence of all three matches will be assigned to `$E` is something that the programmer can specify in the rule itself, using a *search directive*.

4.7.1 Search Directives

Rule Language rules can have one of three execution-directing keywords attached. These keywords are placed directly after the **`begin`** keyword of the rule body, and have the following meanings:

`once.` If there are multiple matches for this rule, then the rule will be executed exactly *once*, for the *first* match encountered.

`continuous.` This keyword is short for *continuous matching*. If there are multiple matches for this rule, then the rule will be executed for each of these matches, in order. The effect is making one 'sweep' with the rule over the domain tree.

`reapplication.` This keyword is short for *exhaustive reapplication*. It repeatedly performs a continuous match, until the rule no longer matches at all.

continuous is the default execution strategy for rules without an explicit keyword.

If we have the following rule:

```
rule R ;
begin once
(any Add any).E [ contains (A Add (B Add C)).SubE ]
→
$E [$SubE = (B Add C) ]
end.
```

and the following domain tree (with two possible matches (for `$SubE`) in bold type):

(A Add **(A Add (B Add C))**) Add (B Add **(A Add (B Add C))**)

then the domain tree will be transformed into:

(A Add (B Add C)) Add (B Add (A Add B Add C))

The same rule, with **continuous** instead of **once** specified, will transform it into:

(A Add (B Add C)) Add (B Add (B Add C))

Finally, if the rule were made reapplicative, it would restart the search after this first sweep, find the match created as a result of the first sweep's action pattern, and transform the tree into:

(B Add C) Add (B Add (B Add C))

4.7.2 Infinite Loops

Continuous matching may cause a rule application to enter an infinite loop, because the action part of each rule application can change or rearrange the domain tree in such a way that new matches keep appearing forever. For example:

```
A Op B
→
C Op (A Op B)
```

When applied continuously, this rule tries to create the 'infinite' tree 'C Op (C Op (C Op ... (C Op (A Op B)) ...))', and the match pattern will never stop matching. As in all programming languages, the compiler cannot always detect this, so it is up to the rule programmer to avoid rules like this.

In contrast, the rule:

```
A Op B
→
A Op B
```

does *not* cause an infinite loop, because continuous matching always *continues*, and never tries to match the **$top** of a previous match again. This rule (as well as any other rule that leaves the parse tree unchanged) would, however, never terminate if applied in **reapplication** mode.

4.7.3 Multiple Matching with 'contains'

In the presence of **contains** clauses, the rules for multiple matches given in the previous section are no longer sufficient. Without **contains**, each of the multiple matches is guaranteed a separate **$top** node. With the **contains**, we can have multiple matches within the same **$top** rooted structure. Consider for instance the following tree:

$(A \text{ Mult } C_1) \text{ Add } (B \text{ Mult } (C_2 \text{ Add } C_3))$

are really all just nodes 'C', indexed for identification purposes. Now consider the following rule:

```
begin continuous
(any operator any).E [ contains C.SomeC ]
→
$E
end.
```

This rule will match 8 times, namely (in matching order):

$E = (A \text{ Mult } C_1) \text{ Add } (B \text{ Mult } (C_2 \text{ Add } C_3))$	$\$SomeC = C_1$
$E = (A \text{ Mult } C_1) \text{ Add } (B \text{ Mult } (C_2 \text{ Add } C_3))$	$\$SomeC = C_2$
$E = (A \text{ Mult } C_1) \text{ Add } (B \text{ Mult } (C_2 \text{ Add } C_3))$	$\$SomeC = C_3$
$E = (A \text{ Mult } C_1)$	$\$SomeC = C_1$
$E = (B \text{ Mult } (C_2 \text{ Add } C_3))$	$\$SomeC = C_2$
$E = (B \text{ Mult } (C_2 \text{ Add } C_3))$	$\$SomeC = C_3$
$E = (C_2 \text{ Add } C_3)$	$\$SomeC = C_2$
$E = (C_2 \text{ Add } C_3)$	$\$SomeC = C_3$

A possible semantics (and a corresponding implementation) of continuous matching might *first* search the domain tree for all these possible matches (like we did for constructing the above table), store these matches in some kind of data structure somewhere, and then process the actions for each of these matches.

The given example clearly shows that this 'snapshot-approach' might not always be a very useful route to take, *if taken literally*: the very first action executed by the example rule replaces the entire domain tree with the node E — how should the remaining seven actions be executed if the tree no longer exists?

The alternative used in the *Rotan* system is to fully execute rules one at a time, and dynamically determine if there still is a 'next' rule or not. The actual implementation issues involved here are complex, but guided by the following considerations:

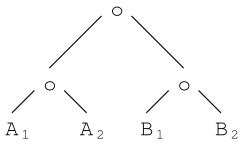
- The domain tree is searched node-left-right for matches.
- Each match can best be thought of as a specific combination of values in a rule variable table, as was done above.
- If a rule uses a **`contains`** then the rule writer must realize that this rule may match several times for the same upper rule variable (see also the example given above). Traditionally, we associate each different match with a different upper rule variable, so it may not be completely obvious that with a **`contains`** we get different matches for one upper rule variable.
- If one rule uses several **`contains`** clauses, then every possible combination of matches for these clauses gives a separate rule match, *for as long as they still exist*. In these combination the leftmost sub-match is most significant, and changes the slowest. Furthermore, if new matches are generated during the matching process, those are actually seen as matches *only if the search has not yet progressed past that point*. This is vital: a search *never backtracks*, but always progresses. The size and form of what is still to be searched may be changed by actions, and for that matter so may the size and form of what has already *been* searched, but the search always 'knows where it is', and progresses from there, left-first through the tree.

The 'leftmost sub-match most significant'-rule given above applies to the search through the *pattern*, not to the search through the domain tree.

As an example of that last rule, consider the match pattern:

```
any [ contains A and contains B ]
```

applied to the tree:



This pattern matches (in order):

A_1	B_1	
A_1	B_2	(if A_1 still exists as a match)
A_2	B_1	(if B_1 still exists as a match)
A_2	B_2	(if B_2 and A_1 still exist as matches)

But if the pattern had been:

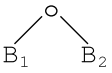
```
any [ contains B and contains A ]
```

then the matches for B and A would have been:

A_1 B_1
A_2 B_1
A_1 B_2
A_2 B_2

in that order (i.e. with B as 'most significant').

We continue with the first match pattern. If, in processing the first match (A_1 B_1), B_1 is replaced by:



we get the new matches:

A_1 B_3
A_1 B_4
A_2 B_3
A_2 B_4

and the match (A_2 B_1) vanishes. If the replacement, however, happens on the *third* original match (A_2 B_1), the search will already have *passed* beyond A_1, and the total list of matches will become:

A_1 B_1
A_1 B_2
A_2 B_1
A_2 B_3
A_2 B_4
A_2 B_2

4.7.4 The Help Keyword

The optional **help** keyword allows a rule programmer to add a descriptive help text to a rule. This text will be shown to the user by the *Rotan* system upon request, or in diagnostic messages from the rule compiler.

The **help** keyword is placed before the **begin** of the rule-body, after the **type** declaration (if any), and followed by a string inside **" "** quotes.

4.8 Drivers and Engines

4.8.1 Drivers

Several rules can be grouped together to form a *driver*. By design, a rule is a unit that accomplishes a micro-task, a driver is a collection of rules that together

perform a logical macro-task, though this logical hierarchy can of course not be enforced by the rule language.

A driver accomplishes its task simply by sequentially executing all the rules that belong to it. The order of these rules is by default the order in which they are encountered in the program file, but this order can be changed by using the **prec** keyword, followed by the names of the rules in the desired order.

Syntax:

```
driver drivername prec <rulenames>
```

4.8.2 Engines

Engines are exactly the same as drivers, except that they consist of a group of drivers (an engine cannot contain rules directly). Each engine can again use the **prec** keyword to specify the execution order.

Typically, an engine will accomplish one large task made up of several drivers.

Syntax:

```
engine enginename prec <drivernames>
```

4.8.3 World

The World is the highest level of hierarchy in the *Rule Language*. There is only one world, and in it a collection of engines can be grouped together (and given an order using **prec**).

Syntax:

```
world prec <enginenames>
```

4.8.4 Rules

In the *Rule Language* implementation each rule specifies in its declaration line what engine and what driver it belongs to, as in:

```
rule R from thisdriver in thatengine ;
```

What follows the **in** keyword is the driver name, what follows the **from** is the engine name.

This scheme does allow us to distribute the rules arbitrarily over any number of files. Whenever rule program files are loaded into the transformation system, the system can easily determine for each rule what engine or driver it belongs to, and execute it in the correct order.

All precedence statements must occur at the beginning of a rule file, but if a project contains multiple rule files, each file can start with its own set of precedence statements (which must still all be unique).

4.8.5 Conclusion

As soon as the *Rule Language* has been instantiated to a specific domain, we possess the tool to implement the third step of Section 3.4.1: create a collection of rule-engines that together implement a specific transformation path on source trees in the language described by the domain.

We will illustrate this in the next chapter with a case study applying to our chosen area of data-parallel compilation, and create a *Rotan* compiler for the intermediate language *Vnus*.

Chapter 5

A *Rotan* Compiler for *Vnus*

5.1 The *Vnus* Language

The *Vnus* language was created at the Delft University of Technology to serve as an intermediate language that would adequately be able to represent data-parallel languages during the compilation phase [Dec98]. It is an architecture-independent language that is easy to parse and primarily intended for machine-consumption, but also has high-level semantics that allow a convenient mapping from various abstractions found in parallel programming languages, such as loops, procedures and functions, communication primitives, and other explicitly parallel constructs.

Vnus also has low-level semantics that allow a formal *calculus of transformations* to be defined in support of the various compilation and optimisation phased. This calculus, called *V-cal* (also described in [Dec98]) allows us to reason about transformations and supply proofs for semantical equivalence of 'before' and 'after' trees.

The following is an example of a simple, implicitly parallel (see Section 5.2) *Vnus* program that initialises four differently-distributed vector arrays, and then calculates the result vector $A = B + C + D$:

```
program


declarations [
    globalvariable A shape [1000000] [block] int,
    globalvariable B shape [1000000] [block] int,
    globalvariable C shape [1000000] [cyclic] int,
    globalvariable D shape [1000000] [blockcyclic 4] int,

    cardinalityvariable i,
    cardinalityvariable j,
]


statements [
```

```
    pragma [independent] forall [j: 1000000] statements [
        assign (A, [j]) j,
        assign (B, [j]) (10, +, j),
        assign (C, [j]) (100, +, j),
        assign (D, [j]) (1000, +, j)
    ],

    pragma [independent] forall [i: 1000000] statements [
        assign (A, [i]) (((B, [i]), +, (C, [i])), +, (D, [i]))
    ],

]
```

This example illustrates a number of the language constructs that make *Vnus* so suitable as an intermediate format for (data-)parallel programs.

- The **`shape`** keyword, which *Vnus* uses to declare array-like data structures.
- The **`block`**, **`cyclic`**, and **`blockcyclic`** keywords signify an explicit data-parallel distribution specification to be applied to the global arrays *A*–*D* (or *shapes* *A*–*D*, in *Vnus*' terminology).
- The **`cardinalityvariable`** keyword signifies that the variable declared that way will be local to a single loop in the program only, and will never have a negative value. Having a special declaration for these variables (that are typically used as array-indices) can aid the compiler in subsequent data-flow analysis.
- The **`pragma`** is the *Vnus* version of *annotations*, or meta-information attached to certain constructs in a program (declarations, statements, and expressions can all be annotated). *Vnus* pragmas are typically directly passed on from annotations specified by the programmer in the higher-level source code (e.g. HPF directives), but can also be the result of analysis performed by the compiler itself.[1]
- The **`independent`** pragma indicates that the loop construct it annotates has no data-dependencies between individual iterations of the loop. These iterations can therefore safely be executed in parallel. Ideally, the compiler would itself be able to deduce when a loop has the 'independent' property (certainly in the case of the simple loops used in the example), but this is not always possible, whereas it often *is* immediately obvious to the programmer who wrote the loop.
- The **`forall`** keyword indicates that the body of the loop is to be executed for every element in its iteration space (parameterised by the value of the cardinality variable), with the individual state changes caused by the iterations

[1] In fact, the *Rotan Vnus* compiler extensively uses internal pragmas to store analysis results obtained during the compilation process.

merged afterwards into a new program state. Changes made to the program state by one iteration will not affect the input state of any other iteration, but if two iterations were to write to the same variable, its final, post-merge value will be unspecified after the **`forall`** is finished.[2] The **`forall`** is one of the most interesting constructs in *Vnus*, as its semantics explicitly allow us to map a number of similar loop constructs found in higher-level parallel programming languages to it (for a more detailed discussion about the subject, see [Dec97a]).

There are also many explicitly parallel language constructs in *Vnus* that we will encounter when we look at the code that the compiler will generate for programs such as the above. These constructs include:

primitives such as for element-wise sending of data,

- The **`forkall`** statement for spanning a program-wide loop that iterates over the processor array.
- The **`send`** and **`receive`** primitives for the element-wise communication of data between processors.
- The **`blocksend`** and **`blockreceive`** primitives for the communication of aggregated data between processors.
- The **`owner`**, **`sender`** and **`isowner`** primitives that provide information derived from the distribution of data elements.
- The **`barrier`** synchronisation statement that pauses execution until all processors have reached the same execution point.

For a more comprehensive explanation of the syntax and semantics of *Vnus*, we refer the reader to [Ree00b], the *Vnus Language Specification*. For ease of reference the full grammar for the *Vnus* language is reproduced in Appendix B.

5.2 Global Design of the Compiler

The combination of a set of transformation rules on a domain, implemented in the *Rule Language* as described in Chapter 4, and the *Rotan* compilation system described in Chapter 3 can be used to create a specific instance of a compiler or optimiser for that domain.

In this chapter, we will show how *Rotan* can be used to build a parallelising, optimising compiler for *Vnus*. Figure 5.1 depicts the components of this system and their interactions.

[2] An **`independent`** pragma can of course be used to explicitly assure the compiler that no such write-dependencies exist between iterations.

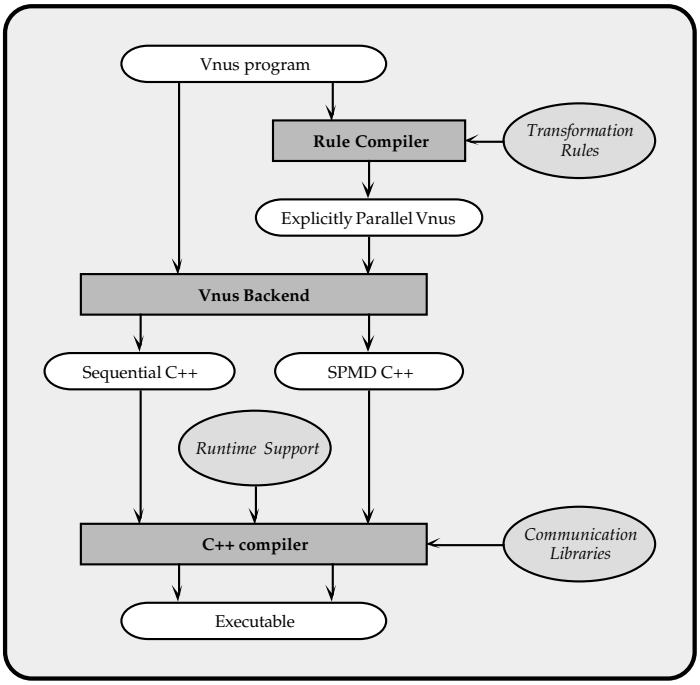


Figure 5.1: *The Vnus Compilation System.*

As mentioned in the previous section, *Vnus* is a rich intermediate language, with a wide range of available constructs and idioms. We distinguish three separate categories of possible *Vnus* programs:

Sequential programs. These do not use any of the parallelism-specific constructs.

Implicitly parallel programs. Here the data structures and statements are annotated with distribution information in the form of *pragmas*.

Explicitly parallel programs. These use the communication and synchronisation primitives present in the *Vnus* language.

The compiler described in this chapter will use implicitly parallel *Vnus* programs as its input. Explicitly parallel *Vnus* programs are its output format.

Before any rules can be written, a *Rotan* domain definition for *Vnus* must be created, as explained in Section 3.4.2. For further reference, this complete domain definition is reproduced in AppendixC.

5.2.1 Sequential Compilation

An implicitly parallel *Vnus* program can be reduced to a sequential one by discarding all the distribution information (i.e. ignoring the pragmas). Presence or absence of such information will, by definition, not change the semantics of the program. It is therefore possible to create a sequential executable for any implicitly parallel *Vnus* program by following the compilation sequence as outlined in the left half of Figure 5.1.

A source program is fed directly to the *Vnus* backend (which exists separately from the *Rotan* system itself), which discards the distribution information, and performs a translation of the resulting sequential *Vnus* program to sequential C++. This C++ program is further processed by a conventional C++ compiler, and linked against a custom runtime support library.

5.2.2 Parallel Compilation

The *Vnus*-instantiated Rule Compiler is used to transform an implicitly parallel *Vnus* program into an explicitly parallel *Vnus* program in SPMD format. The *Vnus* backend again takes care of a translation from SPMD *Vnus* to SPMD C++, that is: it generates C++ code which uses calls to explicit communications and synchronisation routines corresponding to the explicitly parallel primitives found in *Vnus*. The C++ code is compiled by a conventional C++ compiler, and linked against the runtime support library and a communications library such as PVM [Gei94], MPI [Sni96], or possibly a custom, platform-specific library. The resulting 'node executable' will then be ready to run on the parallel architecture in question.

There is a large degree of freedom available to the Rule Compiler. Different transformation schemes can lead to different parallel programs that are all semantically equivalent (i.e. they lead to identical output), but that have varying amounts of efficiency and exploited parallelism. There is not one single 'correct' parallelisation of a *Vnus* program, but there will be translations that are *better* than others, in terms of optimising for a specific evaluation criteria.

There are different metrics one could be interested in optimising for, e.g. memory usage. For the purposes of this thesis, however, there is only a single evaluation function of interest: we wish to minimise the execution time of the program. In the remainder of this chapter we will describe the transformations chosen to achieve that effect in the *Rotan Vnus* compiler.

The implementation of a complete, production-strength compiler for *Vnus* is out of the scope of this thesis. Our interest is in creating a proof-of-concept compiler that can be used to test the concept of a *Rotan*-based compiler. The compiler must still be strong enough to handle more than toy problems. With this in mind we state the following design constraints for the *Rotan Vnus* compiler:

- The compiler should correctly translate a substantial range of *Vnus* programs.

- Further restrictions on the set of supported *Vnus* programs are allowed, provided that it is possible to write a semantically equivalent *Vnus* program that *is* supported. It is not interesting to have to program rules for handling corner cases or sugaring constructs that can be easily rephrased by the programmer.

The *Rotan Vnus* compiler implementation consists of two major transformation phases: a parallelisation phase and an optimisation phase. Each phase consists of a number of engines, each engine being made up of a number of drivers, each driver consisting of a number of individual rules. Engines, drivers and rules are always applied in a specified order.

In the remainder of this chapter we will examine the transformation sequences in more detail.

5.3 The Parallelisation Phase

The parallelisation phase consists of seven engines, divided over two sub-phases:

- **Normalisation and Analysis**

  In this sub-phase, the engines and drivers transform the input program into a version that remains functionally equivalent, but with certain language constructs replaced by others. The goal of this engine is to decrease the syntactical variety of input programs that will be offered to the subsequent parallelisation engines. This allows both the number and the complexity of these rules to be kept down, even though the compiler will still be able to handle a wide range of *Vnus* programs.

  The Normalisation and Analysis sub-phase consists of the following engines:

  1. Preliminary normalisation
  2. Function-to-procedure conversion
  3. Global variable removal
  4. Shape analysis

- **Parallelisation**

  In this sub-phase the actual transformation from a sequential, implicitly parallel *Vnus* program to an explicitly parallel *Vnus* program takes place. The resulting SPMD program will be based on element-wise communication, and therefore as yet be highly inefficient. This will be remedied in the Optimisation sub-phase.

  The Parallelisation sub-phase consists of the following engines:

  1. SPMD insertion

2. Temporaries insertion
3. Owner test insertion

We will further describe the entire parallelisation phase by focusing on each of the seven engines in turn, showing examples of the transformations they implement as well as examples of the rules themselves.

5.3.1 Preliminary Normalisation

This is a catch-all engine for a number of smaller normalisation rules that bring the program into a more consistent, manageable form. After this engine has processed a *Vnus* program, the following postconditions will hold:

- The program contains no variables that are initialised upon definition. All such initialisations are now expressed as explicit assignments to that variable in the corresponding scope code block. This reduces the number of cases subsequent rules need to take into account.

  For example:

```
declarations [
    globalvariable x long 666
]

statements [
    ...
]
```

  becomes:

```
declarations [
    globalvariable x long
]

statements [
    assign x 666,
    ...
]
```

- All the code present in the main body of the program is moved to a new procedure with a fixed name. The remaining main body only consists of a single call to this procedure. Subsequent transformations can now all be written as rules applying to procedure definitions and declarations.

A typical example of a rule used in this engine (the token **&&** is an alternative notation for **and**):

```
RULE pre2 FROM pre IN PreliminaryRules
HELP "Convert main body global variable initializers to explicit assignments"

BEGIN

  Vnusprog.P
```

```
    [
      Declarations CONTAINS DeclGlobalVariable.G
      [
        NOT (Init MATCHES ExprNull) &&
        $INIT = Init &&
        $ID = Name
      ]
      &&
      Statements
      [
        Statements MATCHES
        <
          (...).XX
          (Statement [ NOT CONTAINS FlagPragma [Name == "skipthis"] ]*).SS
        >.STATEMENTS
      ]
    ]
    ->
    Vnusprog.P
    [
      $G = DeclGlobalVariable.G [ Init = ExprNull ]
      &&
      $STATEMENTS =
      <
        $XX
        SmtAssign
        [
          Lhs = LocName [ Name = $IDCLONE ] &&
          Rhs = $INIT &&
          Pragmas = < FlagPragma [ Name = "skipthis" ] >
        ]
        $SS
      >
    ]
    {
          // Embedded C++ code
      $IDCLONE = (String *)$ID->Clone();
    }

  END.
```

The use of the *skipthis* Boolean flag pragma is a typical construct that returns in many rules. By adding this pragma to a matched and processed node, we ensure that the same node (in this example, the *Statement* node `SS`) will not match a second time. In this manner we can force the rule to iterate exactly once over all the statements in a list.

Since almost every node in a *Vnus* program can have a pragma attached, and pragmas are entirely user-defined, pragma lists are a convenient place to use for storing information resulting from analyses rules, or (as seen above) to just serve as a scratch area where temporary or meta-results can be kept by rules to use in coordinating their effects or steering their own traversal of the input tree.

5.3.2 Function-to-procedure Conversion

In *Vnus* both function and procedure calls are allowed. For most of the transformations relevant to this thesis, the distinction between the two is not important, in which case the word *routine* will be used to describe either or both.

Routines can either be linked to from an external library, or defined in the *Vnus* program itself. Routines are parameterised, can be recursive, and can contain references to global variables, but have no further side effects; all state changes take place through the routine's parameters or through changes to the global variables.

The translation of functions and procedures will be in large part similar. In order to avoid having to write a near-duplicate for every rule that performs an action on a procedure, we first execute a conversion engine in which all functions are rewritten to procedures.

For example:

```
formalvariable a FacRec long,

function FacRec [a] long statements FacRec [
  if (a, =, 0l)  statements [
      return 1l
  ] statements [
      return (a, *, functioncall FacRec [(a,-,1l)])
  ]
],

... functioncall FacRec [42] ...
```

becomes:

```
formalvariable a FacRec long,
formalvariable retFacRec FacRec pointer long,
localvariable fcres_1 scope_1 long,

procedure FacRec [a, retFacRec] statements FacRec [
  if (a, =, 0l) statements [
      assign *retFacRec 1l,
      return
  ] statements scope_1 [
      procedurecall FacRec [(a, -, 1l), &fcres_1],
      assign *retFacRec (a, *, fcres_1),
      return
  ]
],

localvariable fcres_2 scope_0 long,

procedurecall FacRec [42, &fcres_2],

... fcres_2 ...
```

Converting functions to procedures is a two-stage process:

- All function *definitions* are rewritten as equivalent procedure definitions. This is done by converting the return variable of the function to an additional parameter[3], which retains the same distribution as the original return variable. The return statement is replaced by an assignment to the new parameter, and the function declaration is replaced by a procedure declaration. The body of the function can be copied across mostly unchanged.

[3]This is implemented using a pointer, as all parameter passing in *Vnus* is by value.

- All function *calls*, both in the main body as well as in the routine bodies themselves, are transformed into equivalent procedure calls. Function calls are allowed in the right hand side of assignment statements and in the expressions that form the arguments to other routine calls. In both cases the treatment is the same: a new temporary variable is introduced, with the same distribution as the original function's return variable. A call to the procedure created in the first step of this engine is added in front of the statement in which the original function call appeared (with the new temporary in the place of the 'return parameter'), and the original function call is replaced by a reference to the temporary variable.

The above approach will lead to correct code in all the 'difficult' cases: multiple calls in one expression, nested calls, even recursive calls in the function body itself. Some care has to be taken concerning the order in which the procedure calls are added, however.

In the case of functions and procedures, we could have considered functions to be a non-essential 'convenience' construct in *Vnus*, and saved effort by simply restricting our set of acceptable *Vnus* programs to those using procedures only, effectively leaving the conversion up to the original programmer. However, this would be too restrictive, particularly since the transformation process in this case is a tedious task for humans to perform, but fairly straightforward to automate. The function-converting rules in question also present a good example of the expressive power of the *Rule Language*.

Like most of the other components of the *Vnus* paralleliser, the function-to-procedure engine focuses on equivalence rather than efficiency. We have not considered it worth the effort to implement certain obvious, well-known optimisations in this engine, such as the merging of the duplicate temporaries that will be the result for handling cases like e.g. $f(x) + f(x)$.

Although after this engine has been applied we are now guaranteed that all our routines are in fact procedures, we will continue to use the word routine wherever the text applies with equal strength to functions as well as to procedures.

The following is a typical example of a rule in this engine:

```
RULE ftp5 FROM ftp IN FunctionsToProcedures
HELP "Convert all uses of function types in declarations to procedure types"

BEGIN

  Vnusprog.P
  [
    Declarations CONTAINS RoutineType.RT
    [
      MATCHES TypeFunction
      [
        $RETURN = Rettype
      ] &&
      Formals MATCHES < (...).PT >
    ]
  ]
  ->
```

```
    Vnusprog.P
    [
      $RT = TypeProcedure
      [
        Formals = < $PT TypePointer [ Elmtype = $RETURN ] >
      ]
    ]
  END.
```

5.3.3 Global Variable Removal

Sequential *Vnus* allows references to global variables from within routine bodies. A characteristic of SPMD code is that global variables are not possible, as there is no shared memory space. All usage of such variables must therefore be replaced by an equivalent scheme that uses only local variables and routine parameters.

This engine performs the following steps:

- All global variable declarations are converted into field declarations within a single new global record, called the *common block*.
- The common block is added as an extra parameter to every single routine definition and routine call in the program (with the exception of external routines, which are guaranteed not to refer to global variables to begin with).
- All references to global variables are replaced by references to fields within the common block.

After this engine has finished, we are guaranteed that (a) the program now only contains a single global variable (the common block), and (b) none of the program's routine code blocks contains any references to global variables any more (the common block is passed as a parameter to every routine). The only reference left to a global variable is the single procedure call in the main body of the program, where the common block is first 'inserted' into the routine call chain. This single global variable occurrence can now be handled by a small number of special-purpose rules in the parallelisation sub-phase.

The following rule from the engine illustrates the use of list manipulation in the *Rule Language*, and the ability to escape to C++ for functionality not provided by the *Rule Language*, in this case for the administrative task of creating a unique identifier.

```
RULE cbp3 FROM cbp IN CommonBlockPropagation
HELP "Add a common block parameter to all procedure declarations"

BEGIN

  Vnusprog.P
  [
    Declarations CONTAINS BlockRoutineDeclaration.BD
    [
      Status != "done" &&
      Parms MATCHES < (...).FORMALS >.FP &&
```

```
      Body MATCHES Block [ $SCOPE = Scope ] &&
      $PROCNAME = Name
    ]
    &&
    Declarations MATCHES < (...).DECLARATIONS >
    &&
    Declarations CONTAINS DeclGlobalVariable.G
    [
      CONTAINS FlagPragma [ Name == "isCommonBlockPtr" ] &&
      $TYPE = T
    ]
  ]
  ->
  Vnusprog.P
  [
    $BD = BlockRoutineDeclaration.BD [ Status = "done" ] &&
    Declarations =
    <
      $DECLARATIONS
      DeclFormalVariable
      [
        Scope = $SCOPE &&
        T = $TYPE &&
        Name = $ID
      ]
    >
    &&
    $FP = < $FORMALS $ID >
  ]
  {
      // Embedded C++ code
    $ID = new String(idGenerator->Unique("cb_"));

    if (*$SCOPE == "no scopename")
    {
      *$SCOPE = new String(idGenerator->Unique("scope_"));
    }
  }

END.
```

5.3.4 Shape Analysis

In this sub-phase, the program tree is traversed by rules that collect information about the program and store it in a format and a location (pragma lists are again used for this) that subsequent rules can easily use.

This engine gathers information about the distribution, size, and type of the arrays used in the program, stores this in a custom *ShapeLocationPragma*, and attaches a reference to this pragma to every occurrence of the shape in question.

In essence, this engine parses the *Vnus* declarations, and constructs a symbol table for the shape identifiers.

This engine is not the only place where analysis occurs during the compilation trail. As the program subsequently passes through other engines, there will be further, mostly local and rule-specific, analyses that need to be done. However, the tree decoration that results from the shape analysis engine is used by practically every other subsequent rule in the compiler.

As an example of the effect of this engine:

```
globalvariable A shape [20] [block] int,

...
assign (A, [0]) 666
```

becomes:

```
globalvariable A shape [20] [block] int,

...
assign (pragma [shapeloc id=A basetype=int dist=block(0)] A, [0]) 666,
```

And an example of one of the central rules in this engine is the following:

```
RULE slp03 FROM slp IN ShapeLocationPragmify
HELP "Attach a ShapeLocationPragma to all uses of global shapes in expressions."

BEGIN

  Vnusprog.P
  [
    Declarations CONTAINS ShapeLocationPragma.SLP
    [
      $ID = Shape
    ]
    &&
    Declarations CONTAINS BlockRoutineDeclaration
    [
      Body CONTAINS ExprName.EN
      [
        Name == $ID &&
        Pragmas MATCHES < (...).PRLIST > &&
        NOT Pragmas CONTAINS ShapeLocationPragma
      ]
    ]
  ]
  ->
  Vnusprog.P
  [
    $EN = ExprName.EN
    [
      Pragmas =
      <
        $SLP
        $PRLIST
      >
    ]
  ]

END.
```

Note that as the default application mode of this rule is 'continuous', a single application of this rule will result in all references to all global shapes in a program being tagged.

5.3.5 SPMD Insertion

Vnus provides the *forkall* iteration construct as a means of specifying that a program is SPMD. Its syntax resembles that of the conventional *for* statement, but there are a number of restrictions and the semantics are quite different.

Like the *for* statement, the *forkall* specifies a cardinality variable with an associated iteration space. The first restriction is that a *forkall* can only specify one single cardinality variable (usually called p). The second restriction is that p always iterates over the range $0..nrOfProcessors$, where *nrOfProcessors* is an externally linked variable whose value is system-dependent. The third restriction is that there may occur only one *forkall* in a *Vnus* program, and that it must be the outermost construct in the main body of the program.

The semantics of the *forkall* in this form are simple: the *forkall* signifies that the program is an SPMD program: *nrOfProcessors* identical processes will be started, each process executing the body of the *forkall*, and therefore parameterised in p, the process number.

The following example shows how the forkall is introduced and the processor parameter p is installed into the common block (the declaration of which has been expanded to accommodate it by one of the earlier rules in the engine):

```
statements [
  assign commonBlockPtr1 &commonBlock1,
  procedurecall rc_main2 [commonBlockPtr1]
]
```

becomes:

```
externalvariable numberOfProcessors int,
cardinalityvariable procnr1

statements [
  forkall [procnr1:numberOfProcessors] statements forkall_scope [
    assign field commonBlock1 _p procnr1,
    assign commonBlockPtr1 &commonBlock1,
    procedurecall rc_main2 [commonBlockPtr1]
  ]
]
```

Introducing SPMD is the first step towards converting an implicitly parallel program into an explicitly parallel one. It is necessary, but not sufficient, to encapsulate the program's main body by a newly created *forkall* construct. Care must also be taken to convert all the remaining global variables (of which there is only one at this point, thanks to our common block engine) occurring in the program into local variables. In an explicitly parallel SPMD context, global variables are not allowed: every process only has its own memory space, and all sharing is done by message-passing.

Our approach towards localising shape A is simple: we give every process a private copy of A, even if A is a distributed shape (in which case the distribution info is also copied). For distributed shapes it is a conventional and elegant approach to perform *shrinking*, and provide each process not with the entire shape A but with a derived local shape $A(p)$, containing only those elements of A which will actually reside with process p. In the program code itself all index expressions into A would then have to be encapsulated in *global-to-local* function calls. While this transformation may have some effect on the actual performance of

the resulting code (e.g. by avoiding cache or other memory-related slow-downs for large arrays), these effects would be small for the majority of examples used throughout this thesis. We have therefore decided not to implement shrinking in our *Vnus Rotan* compiler.

5.3.6 Temporaries Insertion

The basic communication scheme the *Rotan Vnus* compiler implements when parallelising has been described in Section 2.4.2.

Communication insertion forms the heart of the *Vnus* compiler. For each processor, all non-local data that is needed by the computations must be retrieved from or sent to other processes, while maintaining program correctness.

With respect to the communication primitives this scheme uses the non-blocking sends and blocking receives supported by *Vnus*. As far as computations are concerned, the *owner-computes* rule is used.

In the case of a data element having more than one owner (as is the case for replicated shapes), all owners will execute the necessary code to keep their copy of the element current, but we assume that only (and exactly) one of these owners is designated the *Sender*, responsible for corresponding the element's value to other processes. The sender is chosen according to some *sender-determination* function. In advanced implementations, the sender-determination function can, for instance, be based on the topology of the network. In our case we settle for a simpler choice, choosing index 0 in every replicated dimension. Thus, process 0 itself is always responsible for all totally replicated data. We have also assigned it the task of performing I/O and other interfacing with the outside world that should not be performed by all processes.

In *Vnus*, all assignments only involve scalars. With this in mind, we introduce a scheme based on scalar or *element-wise* communication. Aggregating these element-wise sends and receives into block sends and block receives for better performance is a task left to one of the optimisation engines, although in the current phase we will also take care of some elementary data analysis in order to pave the way.

Each assignment statement is split into a number of *communication* assignments followed by one *computation* assignment. For each data element referenced in the right hand side of the original statement, a local temporary scalar variable is introduced, which is assigned to in the communication assignment. The computation assignment can subsequently be executed completely locally (because owner computes hold, the result of the computation is always local as well).

Creating temporaries also needs to be done for other 'right hand side' contexts, such as condition expressions for *if* and *while* statements, and the actual arguments of routine calls.

Assignment statements where the right-hand side consists of a single non-local data element, however, are not affected, but are marked straight away as communication assignments.

For example:

```
assign (A, [i]) (((B, [i]), +, (C, [i])), +, (D, [i]))
```

becomes:

```
pragma [communication] assign tmp_it1_0 (B, [i]),
pragma [communication] assign tmp_it1_1 (C, [i]),
pragma [communication] assign tmp_it1_2 (D, [i]),
pragma [computation]
    assign (A, [i]) ((tmp_it1_0, +, tmp_it1_1), +, tmp_it1_2)
```

5.3.7 Owner Test Insertion

This engine implements the final steps in the introduction of the communication scheme: rewriting both the communication assignments and the computation assignments to explicitly parallel versions.

Computation assignments need to be encapsulated by an owner-test, so that the owner-computes rule is are actually implemented. Computation assignments need only be performed by the owner(s) of the element referenced in the left hand side of the statement.

Communication assignments are expanded into send/receive pairs. The element is sent only if p is the Sender for that element, the element is received by each p that is an owner for the left hand side in question.[4]

As mentioned before, this communication scheme is extremely inefficient, not only because of the element-wise sending, but also because we do not avoid same-processor (or *send-to-self)* communications, which would happen if the source and destination processors for a data element for a processor turn out to be one and the same.

While send-to-self elimination rules could be implemented in the rule language, we chose not to do so because it is already automatically taken care of at a lower level by the runtime communication libraries. This has the added advantage of being a run-time solution that also works in those cases where it is not possible to determine at compile time whether a communication would be a send-to-self or not. A disadvantage, however, is that the communication calls are still generated, and can in certain cases lead to so much overhead (even though no communication will actually take place) that it negatively affects the performance of the program. And example of this will be seen in Section 6.3.2.

The following example shows how communication and computation statements are handled by the owner test insertion engine:

```
pragma [communication] assign tmp_it1_2 (D, [i]),
pragma [computation]
    assign (A, [i]) ((tmp_it1_0, +, tmp_it1_1), +, tmp_it1_2)
```

becomes:

[4] A *send* command can have multiple destinations encoded in the processor number.

```
if (sender (D, [i]), =, field *cb_1 _p) statements [
  if (owner tmp_it1_2, <>, field *cb_1 _p) statements [
    send owner tmp_it1_2 (D, [i])
  ] statements []
] statements [],
if isowner tmp_it1_2 field *cb_1 _p statements [
  if (sender (D, [i]), <>, field *cb_1 _p) statements [
    receive sender (D, [i]) tmp_it1_2
  ] statements [
    assign tmp_it1_2 (D, [i])
  ]
] statements [],
waitpending,
if isowner (A, [i]) field *cb_1 _p statements [
  assign (A, [i]) ((tmp_it1_0, +, tmp_it1_1), +, tmp_it1_2)
] statements []
```

The expression `field *cb_1 _p` retrieves the processor number from the 'common block' record where it was stored previously.

5.4 The Optimisation Phase

Currently, the optimisation phase consists of three engines each implementing a different, efficiency-improving transformation on the original program.

They were chosen because they illustrate a wide spectrum of possible transformations that can be implemented using the *Rule Language*, and because together they are already sufficient to generate parallel programs that run very efficiently (as we will describe in detail in Chapter 6).

The three engines are:

- Communication Aggregation
- Owner Test Absorption
- Owner Test Inlining

We will describe the three optimisations in more detail.

5.4.1 Communication Aggregation

Communication aggregation is an optimisation that searches the input program for occurrences of element-wise communication in a loop context. It then replaces this communication by code which uses a similar loop to fill a local memory buffer instead. This entire buffer of values is then communicated at once by a so-called *blocksend* command.

This engine consists of the following sub-engines:

Recognition and lifting. Identifies and tags the loops that this engine can be applied to, and if necessary splits the loop so that code that is not involved (e.g. computations) will remain separate.

Scalar expansion. Recognises and converts uses of scalar temporaries into uses of array elements. The array in question is a newly created local buffer.

Aggregation. Inserts the aggregation template into the code tree, taking care of sending from and receiving into the local buffer.

Cleanup. Ensures that new cardinality variables (e.g. introduced when loops were cloned during the lifting phase) are indeed unique, as required by *Vnus*.

As an example, the large rule that implements the aggregation sub-engine is listed in its entirety in Appendix D.

5.4.2 Owner Test Absorption

Because of the owner-computes rule and because of the fact that only the owner of a data element is responsible for communicating its value to other processors, it follows that in an unoptimised *Vnus* program, loop/if constructs similar to the following will occur many times in many different forms:

```
foreach [i:n] statements [
  if isowner (A, [i]) field *cb_7 _p statements [
    assign (A, [i]) 666
  ] statements []
],
```

For large n, these runtime element-wise tests on ownership rapidly become prohibitively expensive. If the distribution of the shape is known, however, it has been shown in [Ree96] that in many cases the exact range of values that the index i assumes can be statically generated rather than dynamically tested during runtime.

The owner absorption engine implements this conversion, and will transform the above example — assuming a block distribution for shape A — into:

```
procedurecall vnus_blus [getblocksize A 0, numberOfProcessors, 1, 0, n,
                         &blus_n_u00, &blus_j0_low0, field *cb_7 _p],
foreach [u10:blus_n_u00] statements [
  assign (A, [((1, *, u10), +, blus_j0_low0)]) 666
],
```

The utility routine *vnus_blus* calculates the appropriate upperbound and offset (`blus_n_u00` resp. `blus_j0_low0`) for a loop that defines the exact elements of A being accessed. There are similar routines (and accompanying rules) for shapes that have cyclic or block-cyclic distribution.

5.4.3 Owner Test Inlining

In any explicitly parallel *Vnus* program, there will be many calls to the *Owner*, *Sender*, and *IsOwner* primitives. These will be translated by the *Vnus* backend into calls to utility functions, which can again be very expensive.

It is therefore preferable to use these primitives as little as possible. The Owner Test Inlining engine is a collection of small rules that search the input tree for occurrences of these primitives, and replace them (based on the known distributions of the shapes they apply to) with actual expressions (or in some cases, even constants).

The following rule is a typical example of the rules in this engine:

```
RULE io22 FROM io IN InlineOwners
HELP "Inline all Owner calls for cyclic distributed 1d shapes"

BEGIN CONT

  Vnusprog.P
  [
    CONTAINS ExprOwner.E
    [
      Shape MATCHES LocSelection
      [
        $SHAPE = Shape &&
        Shape CONTAINS String.NAME &&
        Indices MATCHES < Expression.EXPR ... >
      ] &&
      CONTAINS ShapeLocationPragma
      [
        Shape == $NAME && Dist MATCHES DistCyclic && Distdim == "0"
      ]
    ]
  ]
  ->
  Vnusprog.P
  [
    $E = ExprBinop
    [
      Optor = OpMod &&
      Operanda = $EXPR &&
      Operandb = ExprName [ Name = "numberOfProcessors" ]
    ]
  ]

END.
```

This rule will have the effect of changing code such as the following (assuming a cyclic distribution for shape B):

```
if isowner (B, [i]) field *cb_13 _p statements [
  assign (B, [i]) 666
] statements []
```

into:

```
if ((i, mod, numberOfProcessors), =, field *cb_13 _p) statements [
  assign (B, [i]) 666
] statements []
```

5.4.4 Conclusion

With the optimisation engines we complete the third step of Section 3.4.1 and have used the *Rotan* system to created a dedicated *Vnus* compiler/optimiser that

can be used in batch mode to transform source code. What now remains is the investigation of the results of this exercise, both in terms of quality of the produced code, and as an evaluation of the overall usability of the *Rotan* system, which we will devote the final two chapters of this thesis to.

Chapter 6

Experimental Results

In Chapter 3 we introduced the core *Rotan* system, and in Chapter 4 we described the *Rule Language* that is used to instantiate a *Rotan* compiler for a particular domain. In Chapter 5 we used the *Rule Language* to create a parallelisation and optimisation engine for *Vnus*.

In this chapter, we will investigate the performance of the *Rotan Vnus* compiler in general, and the parallelising engines in particular, by running various benchmarks and comparisons on the generated executables.

The *Rotan Vnus* compiler consists of two phases. The first is the transformation phase that accepts data-parallel *Vnus* and outputs message-based, explicitly parallel *Vnus*. The second is the optimisation phase that applies three major optimisation engines (communication aggregation, ownertest absorption and ownertest inlining) to the source program.

The *Vnus* backend then maps the final code to C++, which is compiled by a conventional compiler (*gcc* version 3.2) and linked with the run-time system and the communications library (MPI, *mpich* version 1.2.4) into an executable.

For our benchmarks we will use two different algorithms: a matrix multiplication and a successive overrelaxation (SOR). The cores of these algorithms feature examples of the kind of vector/matrix-based numerical operations, such as inproduct calculation and filter application, that are frequently found in the large scientific programs that are typical parallelisation candidates.

Our claim is that *Rotan* is a system that is mature and powerful enough to tackle such real-world problems. In order to support that claim, we will investigate the performance figures that result from using the *Rotan Vnus* compiler to generate parallel versions of these algorithms.

As we are also interested in the contributions of each of our optimisation engines to the overall efficiency of our compiled programs, we start this chapter with a look at how the presence or absence of these engines affects performance

(Section 6.2).

We continue by investigating the effects of different data distributions on the performance, and how well such differences are handled by our system (Section 6.3).

We end the chapter by comparing our results to that of more professional, production-strength parallelising compilers (Section 6.4). Specifically, we will look at the *Timber* compiler for the *Spar/Java* language, which also uses *Vnus* as an intermediate language, and uses the same communication templates as the *Rotan Vnus* compiler. As a second touchstone, we will use the PGHPF High Performance Fortran compiler as an example of a commercial, well-regarded compiler for high performance, parallel computing.

6.1 Matrix Multiplication

The benchmarks and analyses in this section all apply to an implementation of a standard matrix multiplication algorithm $C = A \times B$. The core code for the *Vnus* implementation of this algorithm (presented here slightly simplified for readability, and as yet without the actual data distribution specifications) is as follows:

```
program

declarations [
    globalvariable A shape [1024, 1024] double,
    globalvariable B shape [1024, 1024] double,
    globalvariable C shape [1024, 1024] double,
]

statements [

        // Initialisation
    pragma [independent] foreach [i: 1024] statements [
        pragma [independent] foreach [j: 1024] statements [
            assign (A, [i,j]) 1.0d
            assign (B, [i,j]) 1.0d
            assign (C, [i,j]) 0.0d
        ],
    ],

        // C = A x B
    foreach [k: 1024] statements [
        pragma [independent] foreach [i: 1024] statements [
            pragma [independent] foreach [j: 1024] statements [
                assign (C, [i,j])
                       ((C, [i,j]), +, ((A,[i,k]), *, (B, [k,j]))),
            ],
        ],
    ],
]
```

Some notes about this algorithm and the variants used in the following sections:

- In our examples, matrix sizes are 1024×1024, unless specified otherwise (a smaller size is sometimes used when the resulting execution times would

become impractically large).

- Matrix multiplication involves the repeated calculation of vector inproducts. In our examples, the summation or reduction step of the inproduct calculation is not itself parallelised (which would also be a possibility). Rather, parallelism is achieved by distributing entire inproduct calculations (between a rows of A and columns of B) over the available processors. Each inproduct is then calculated sequentially and locally (i.e. without further need for communication), on individual processors.

- Parallelism is obtained by distributing the matrix C in the first dimension, letting each processor take responsibility for computing one or more entire rows of the result matrix (by virtue of the *owner computes* rule explained in Section 2.4.4). The overall efficiency of the program will depend in large part on how matrices A and B in turn are distributed and how much communication those distributions will cause. It is worth noting that all the elements of B will always be needed by every processor, regardless of the distribution of C.

Some notes about the benchmarks:

- All benchmarks were performed on the ASCI DAS-2 supercomputer cluster located at the Vrije Universiteit Amsterdam. The DAS-2 consists of 72 Dual Pentium-III 1 GHz nodes that communicate through a *Myrinet-2000* high-speed interconnection network. Each node has 2 Gb of memory available. In the benchmarks, only one of the two processors on each node is actually used. Each node is always entirely dedicated to the executable running on it.

- All execution times shown in the tables and graphs in this chapter are measured in seconds, and denote the actual time spent on the program's `main` routine. The overhead associated with e.g. distributing the programs to the individual nodes is not included in these numbers.

- Each parallel program is executed three times. For each run, the *longest* execution time of all nodes is used as 'the' amount of time the program takes: a parallel program is after all only as fast as its slowest component. For the three runs, in turn, the largest value is chosen as the final execution time to be plotted and used in the graphs. The minimum and average execution times are similarly recorded in the tables for comparison purposes. The bottom row of the time tables is reserved for timings obtained running the sequential version of the program 16 times.

6.2 The Effect of the Optimisation Engines

To study the effect of our three major engines, we measure the execution times of our matrix multiplication program with various combinations of the engines switched on and off.

The benchmarks in this section were performed with three matrices distributed as follows:

```
A shape [256, 256] [block, collapsed] double,
B shape [256, 256] [collapsed, cyclic] double,
C shape [256, 256] [block, collapsed] double,
```

The smaller matrix sizes ($n = 256$) are necessary because with one or more rule engines switched off the execution times will often increase with orders of magnitude.

The distribution of the matrices is intended to be realistic in terms of what can be expected if a matrix multiplication were to be a part of a lager program and context. Important is that a significant amount of communication will need to take place. There are certainly more efficient distribution combinations possible; even some that would lead to absolutely no communication at all (as described in section 6.3). For investigating the effect of the optimisation engines, however, that is not a desired property.

6.2.1 All Optimisations Off

Optimising rule engines:

```
communication aggregation: off
ownertest absorption:      off
ownerfunc inlining:        off
```

#p	min	avg	max
1	138.3	138.4	**138.4**
2	151.1	151.2	**151.3**
4	501.5	513.8	**537.4**
8	476.3	480.1	**484.4**
16	328.2	328.8	**329.4**
32	245.0	245.3	**245.7**
64	191.2	191.4	**191.7**
seq	1.0	1.1	**1.3**

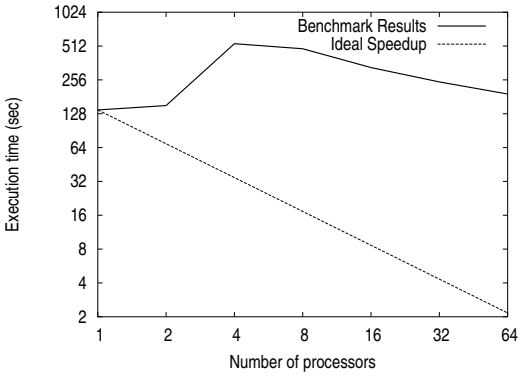


Figure 6.1: *Matrix multiplication; all optimisations off.*

Analysis: With all optimisation engines turned off, the program will dynamically execute, for each processor, $O(n^3)$ element-wise *send/receive* function calls

for accesses to B, $O(n^3)$ local copy operations for accesses to A and C, and $O(n^3)$ *owner* and *sender* calls for all three matrices.

This large amount of element-wise communication is, especially for small numbers of processors, outweighing any speedup gained by the distribution of the computation.

The sequential execution for this program does not take more than 1.1 seconds, which is two orders of magnitude faster than even the best parallel case. It is clear that the naive, element-wise communication scheme is, as expected, expensive to the point of being of no value without further optimisation.

6.2.2 Ownerfunctions Inlined

Optimising rule engines:

```
communication aggregation: off
ownertest absorption:      off
ownerfunc inlining:        on
```

#p	min	avg	max
1	12.3	12.3	**12.3**
2	82.8	82.9	**82.9**
4	290.8	294.8	**300.5**
8	345.4	348.8	**351.2**
16	194.1	195.1	**195.9**
32	112.2	112.5	**112.7**
64	61.3	61.4	**61.6**
seq	1.0	1.1	**1.3**

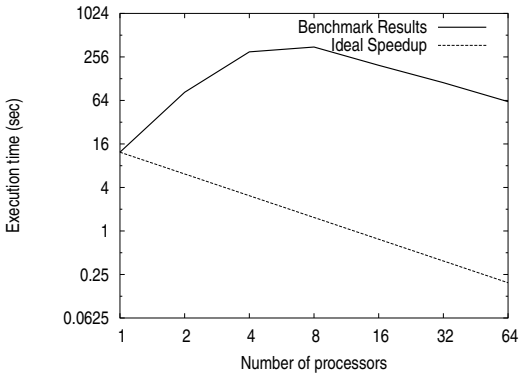


Figure 6.2: *Matrix multiplication; ownerfunctions inlined.*

The curve has the same general shape as in the previous example, which is to be expected, as the generated amount of communication and local copying will still be $O(n^3)$. The significant change here is that now the *owner/sender* function calls have been replaced by direct expressions, and no longer contribute to the overhead.

As a result, the absolute execution times of this benchmark are improved by as much as a full order of magnitude, especially for small numbers of processors (where the amount of actual computation per processor still outweighs the amount of communication).

The execution time for the $n = 1$ case is now only 11 rather than 140 times slower than the sequential version of the program.

Ownerfunction Inlining is not a parallelising optimisation as such, but it is clearly an important step towards getting acceptable performance out of any parallel *Vnus* program to begin with.

6.2.3 Ownertests Absorbed

Optimising rule engines:

```
communication aggregation: off
ownertest absorption:      on
ownerfunc inlining:        off
```

#p	min	avg	max
1	128.3	128.4	**128.5**
2	145.3	145.3	**145.4**
4	454.2	462.4	**470.1**
8	474.2	477.2	**480.6**
16	328.9	329.4	**329.9**
32	245.0	245.4	**245.8**
64	192.2	192.4	**192.6**
seq	1.0	1.1	**1.3**

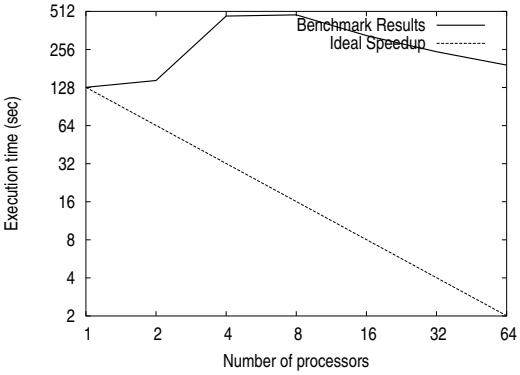


Figure 6.3: *Matrix multiplication; ownertests absorbed.*

With Ownerfunction Inlining no longer on, the absolute execution times go back up to the earlier level seen in Section 6.2.1.

With Ownertest Absorption applied, only the initialisation loops (which do not need communication, because they only feature literal constants in their right-hand side) are optimised. These loops will no longer cause traversal and testing the entire index space.

However, the main computation loop, with the array elements used in the right-hand side, will still lead to the same inefficient element-wise communication scheme as before.

The optimised initialisation loops explain the small improvement in execution times for the smaller values of p, but as p grows the effect of these optimisations will become smaller for each processor, while the communication needs for the main loop remain the same: no matter how few rows of A a processor is responsible for, *all* elements of B need to be retrieved from the other processors. It is clear that overall the element-wise communication remains dominant.

6.2.4 Communication Aggregated

Optimising rule engines:

```
communication aggregation: on
ownertest absorption:      off
ownerfunc inlining:        off
```

#p	min	avg	max
1	80.4	80.4	**80.4**
2	178.6	178.6	**178.7**
4	356.8	482.8	**548.8**
8	656.6	657.0	**657.4**
16	1278.3	1278.8	**1279.1**
seq	1.0	1.1	**1.3**

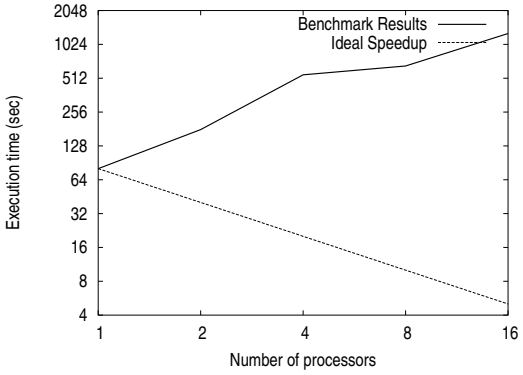


Figure 6.4: *Matrix multiplication; communication aggregated.*

The results for this benchmark clearly show that Communication Aggregation as a stand-alone optimisation does not work. The element-wise communications have been eliminated in favour of buffered sending, but the cure is worse than the disease.

Without owner-test absorption turned on, the newly introduced traversals and tests of the index space that are necessary for the packing and unpacking of elements into their communications buffers theoretically increase the execution time of the program by a factor p. The only case in which this does not lead to decreased performance, is when $p = 1$. For all other values of p, that what is gained in terms of reduced calls to *send/receive* is lost in increased calls to *owner/sender* tests.

6.2.5 All Optimisations On

Optimising rule engines:

```
communication aggregation: on
ownertest absorption:      on
ownerfunc inlining:        on
```

The results for this configuration are shown in Figure 6.5.

Analysis: With all optimisations turned on, we achieve speedup for the first time.

In the $p = 1$ case, the algorithm runs only 3.5 times slower than in the sequential case (3.9 vs 1.1 sec).

For large p the execution times now are so small that measurement errors become significant with respect to the actual execution time.

#p	min	avg	max
1	3.7	3.8	**3.9**
2	2.9	2.9	**2.9**
4	2.0	2.3	**2.5**
8	0.6	0.7	**0.7**
16	0.3	0.3	**0.3**
32	0.2	0.2	**0.2**
64	0.3	0.3	**0.3**
seq	1.0	1.1	**1.3**

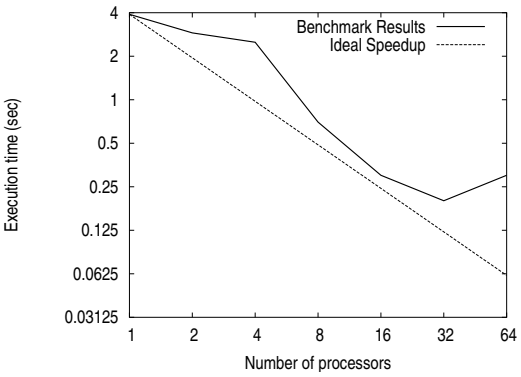


Figure 6.5: *Matrix multiplication; all optimisations on.*

We therefore increase the matrix size n to 1024, and rerun the same benchmark.

6.2.6 Increasing *n* from 256 to 1024

Distribution of matrices:

```
A shape [1024, 1024] [block, collapsed] double,
B shape [1024, 1024] [collapsed, cyclic] double,
C shape [1024, 1024] [block, collapsed] double,

All optimisations on
```

#p	min	avg	max
1	271.9	278.9	**291.5**
2	196.4	203.0	**213.8**
4	189.4	196.4	**200.4**
8	97.3	105.9	**117.1**
16	48.4	53.8	**56.7**
32	15.5	16.0	**16.3**
64	8.9	9.0	**9.0**
seq	66.8	71.2	**81.6**

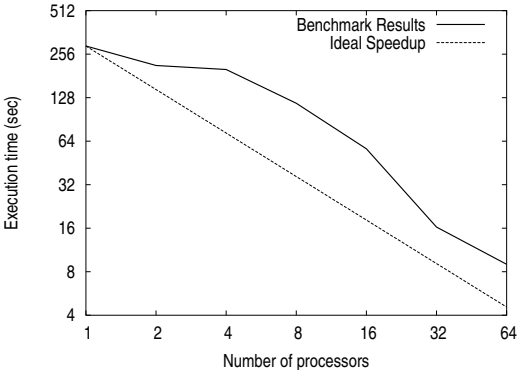


Figure 6.6: *Matrix multiplication;* $n = 1024$.

Analysis: The increased ratio of communication vs computation combined with less noise in the measurements indeed result in a curve that is not as steep, but also less spiky than in the previous benchmark.

With n now 4 times larger, the sequential version of the program takes about 70 seconds, which is indeed the expected $4^3 = 64$ times longer than the 1.1 seconds it took for the smaller matrix size.

The parallel versions show more variation, taking approximately 74 times longer for $p = 1$ and $p = 2$ but increasing to as much as 167 times longer for larger p. The speedup effect remains, but is now less pronounced.

Note that the cyclic distribution of matrix B in the second dimension remains a good one as far as optimising communication is concerned. Because it each column of B will be allocated in its entirety to a single processor, it can therefore be efficiently communication-aggregated, i.e. requiring only a single *blocksend/blockreceive* pair, rather than the multiple ones that would have been necessary had the column been divided over many different processors.

Nevertheless, there are other distribution options, which we will investigate in the next section.

6.3 The Effect of Array Distributions

In the next few examples we will investigate the effect of varying array distributions on the execution time of the program.

All the examples in this section use identical rule sets, with all optimisation engines turned on.

6.3.1 *A* and *C* Block Distributed, *B* Replicated

Distribution of matrices:

```
A shape [1024, 1024] [block, collapsed] double,
B shape [1024, 1024] [collapsed, replicated] double,
C shape [1024, 1024] [block, collapsed] double,
```

#p	min	avg	max
1	231.6	236.9	**239.6**
2	94.2	103.0	**120.4**
4	47.5	51.5	**60.4**
8	23.6	25.3	**27.5**
16	12.3	13.0	**14.7**
32	6.6	6.9	**7.4**
64	3.5	3.7	**3.8**
seq	67.1	71.2	**81.6**

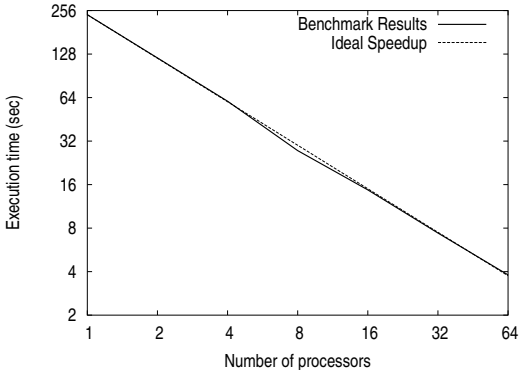


Figure 6.7: *Matrix multiplication; replicated B.*

Analysis: An obvious way in which different distributions can be used to obtain a better-performing matrix multiplication (assuming sufficient memory is available on the individual nodes) is to replicate the entire matrix B over all processors, since we have already seen that every processor needs all of B's elements.

The rule compiler engines will recognise that B is replicated and will generate no communication statements for accesses into that array.

For A and C communication will still be generated, but since they have identical distribution, a good parallelisation of this version does not need to use any communication between processors at all: the rows of A and C are always guaranteed to reside on the same processor.

There is currently no rule engine in the *Rotan Vnus* compiler that implements this *Send-to-self Elimination* for A and C, but the *Vnus* run-time system in fact implements it at run-time in the software layer that lies between the *Vnus* and the MPI communication primitives.

Therefore, even though the SPMD code generated by the rule compiler will still contain *send* and *receive* function calls for the accesses to elements of A and C, these are executed by the run-time system as entirely local buffer copy operations, thus leading to a program that performs no actual communication at all. All that remains is the overhead of the unnecessary function calls.

This explains why the results for this program so closely follow the ideal speedup curve. There is no communication, and the use of the *Owner-test Absorption* engine ensures that the computation is divided over the processors without overhead or redundancy.

6.3.2 *A* and *C* Cyclic Distributed, *B* Replicated

Distribution of matrices:

```
A shape [1024, 1024] [cyclic, collapsed] double,
B shape [1024, 1024] [collapsed, replicated] double,
C shape [1024, 1024] [cyclic, collapsed] double,
```

Analysis: In theory this graph should have been identical to the one in Section 6.3.1, where A and C were block distributed, because the send-to-selfs get eliminated by the run-time system in exactly the same way. Instead the program runs about 10–20% slower, although the linearity of the speedup is preserved.

The difference can be explained by looking at the matrix multiplication algorithm used and realising that the cyclic distributions of A and C are not well-suited to the row-based inner computation loop. For such a loop, the accesses into these arrays will mostly be non-contiguous. This means that the processor caches will not be utilised as well as they can be in the block distributed case. The resulting overhead account for the slowdown of the program.

In both cases, the overhead of the communication-related statements and calls, even if they do not actually lead to communication, also contributes to the absolute execution times. It is this kind of overhead that can be eliminated by

#p	min	avg	max
1	208.4	227.6	**237.3**
2	109.3	123.9	**134.5**
4	56.2	62.2	**67.8**
8	29.1	31.5	**34.4**
16	14.8	16.2	**17.7**
32	8.0	8.5	**9.2**
64	4.3	4.5	**4.8**
seq	66.9	71.0	**81.5**

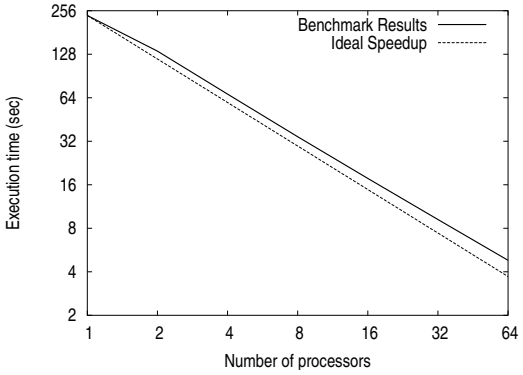


Figure 6.8: *Matrix multiplication; A and C cyclic.*

the application of the *shrinking* optimisation described earlier, at the expense of having to do more computation in the index space.

6.3.3 *C* Block Distributed, *A* and *B* Replicated

Distribution of matrices:

```
A shape [1024, 1024] [collapsed, replicated] double,
B shape [1024, 1024] [collapsed, replicated] double,
C shape [1024, 1024] [block, collapsed] double,
```

#p	min	avg	max
1	145.7	154.3	**165.4**
2	74.3	78.6	**83.1**
4	36.8	38.5	**41.6**
8	18.3	19.4	**21.1**
16	8.9	9.6	**10.8**
32	4.4	5.1	**5.6**
64	1.9	2.0	**2.3**
seq	66.9	70.9	**81.6**

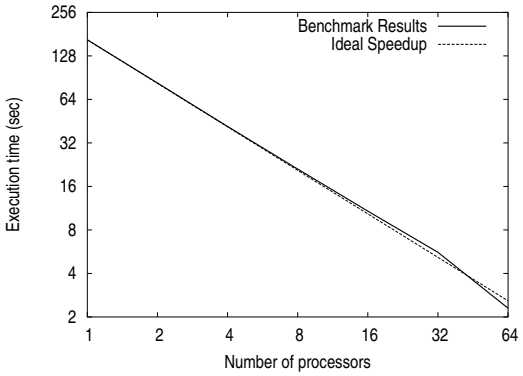


Figure 6.9: *Matrix multiplication; A and B replicated.*

Analysis: By returning C to block distribution and replicating A as well as B, we indeed get rid of the slowdown factor, and are now almost entirely aligned with the linear speedup curve.

The superlinear speedup at $p = 64$ is are caused by cache-effects coming into play.

6.3.4 The Effect of Bounds-checking

By default, the *Vnus* language performs bounds checking on all array accesses. This checking can be turned off for an additional speed increase by specifying a global pragma in the source program. We have not done this for the programs benchmarked in this chapter, as it would be an arbitrary backend optimisation not corresponding to typical real world use. It is also not relevant to the issue of how well the optimisation rules are performing. It is nevertheless interesting to see to what extent the overhead associated with this bounds-checking contributes to the program execution time.

The previous benchmark rerun with bounds-checking switched off led to the following results:

Distribution of matrices:

```
A shape [1024, 1024] [collapsed, replicated] double,
B shape [1024, 1024] [collapsed, replicated] double,
C shape [1024, 1024] [block, collapsed] double,

Boundschecking off
All optimisations on
```

#p	min	avg	max
1	121.8	130.3	**139.7**
2	61.2	66.4	**70.3**
4	31.1	33.6	**35.4**
8	14.3	16.5	**17.8**
16	7.4	8.2	**9.2**
32	3.9	4.5	**4.8**
64	1.6	1.7	**1.9**
seq	58.8	62.8	**72.8**

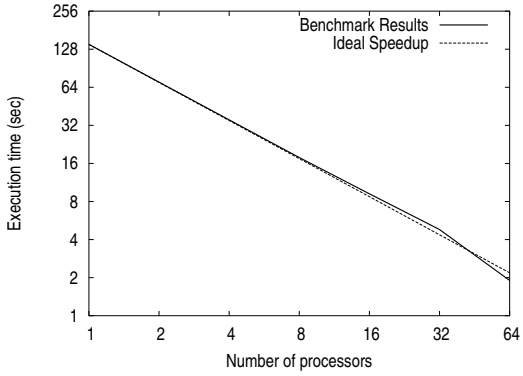


Figure 6.10: *Matrix multiplication; no bounds-checking.*

Analysis: Without bounds-checking, the parallel program consistently executes 15% faster in all cases. A slightly smaller percentage holds for the sequential version of the program, which now only takes 72 seconds instead of 82.

Under these circumstances, the parallel version for $p = 1$ takes only 2 times as long as the sequential program. There is clearly still room for reducing the overhead caused by e.g. the communication administration, but for our prototype compiler the ratio is quite acceptable.

6.4 Comparison with Other Compilers

We have shown in the previous sections that as far as the creation of efficient parallel programs is concerned, the results for our compiler are satisfactory and according to predictions: given the right distributions we see expected levels of speedup.

The *Rotan Vnus* compiler is a proof-of-concept implementation intended to show that it is possible to create a non-trivial parallelising compiler using the tools and infrastructure provided by the *Rotan* system. It is not intended to be feature-complete or of production-level strength.

Nevertheless, it is interesting to compare how well the *Rotan Vnus* compiler performs in comparison with other, less experimental parallelising compilers.

6.4.1 *Timber*

As one example of a more complete and polished parallelising compiler, we will use the *Timber* compiler for *Spar/Java*. This is a particularly appropriate touchstone, because this compiler also uses *Vnus* (albeit a later, more advanced version) as its intermediate format, and because it uses comparable parallelisation templates and optimisation engines, implemented using *Tm* treewalkers (see also section 3.3).

Our experiments use the experimental version 2.0 of the *Timber* system, configured for MPI *mpich* support, and also running on the DAS-2 supercomputer.

We begin by investigating the case where no actual communication is present, i.e. with the distributions equal to the ones used in *Vnus* in Section 6.3.3:

Distribution of matrices:

```
A shape [1024, 1024] [collapsed, replicated] double,
B shape [1024, 1024] [collapsed, replicated] double,
C shape [1024, 1024] [block, collapsed] double,

All optimisations on.
```

The *Spar/Java* program that corresponds to this *Vnus* program is (edited lightly for readability) is:

```
public class matmat
{
    static final int N = 1024;
    static final int bsize =
                  N / spar.lang.DataParallel.getNumberOfProcessors();

    static double A[*,*]
        <$on = (lambda (i j) P[_all])$> = new double[N,N];
    static double B[*,*]
        <$on = (lambda (i j) P[_all])$> = new double[N,N];
    static double C[*,*]
        <$on = (lambda (i j) P[(block i @bsize)])$> = new double[N,N];

    public static void main()
    {
        <$independent$> foreach (i :- 0:N, j :- 0:N)
        {
             A[i,j] = 1.0d;
```

```
                B[i,j] = 1.0d;
                C[i,j] = 0;
            }

            foreach (k:-0:N)
                <$independent$> foreach (i :- 0:N, j :- 0:N)
                {
                    C[i,j] += A[i,k] * B[k,j];
                }
        }
    }
```

As mentioned, although the *Timber* compiler also uses *Vnus* as an intermediate language, its backend is significantly different from the one used for *Rotan*. The *Vnus* to C++ mapper, the *Vnus* format itself, and the run-time system are all improved versions that can be expected to contribute to superior execution times.

In particular, the improvements to the *Vnus* format make it possible to have optimisation engines perform better analyses on the distributed arrays, and generate improved code. The C++ data structures and access methods in the run-time system were also rewritten, specifically in order to speed up array accesses and eliminate administrative overhead.

Adapting *Rotan* to use this newer backend, while possible in principle, was a task that was outside of the scope of this thesis. It should also be noted that the newer backend is not a finished product, but itself a research project in constant flux.

The results of the benchmark for the *Spar/Java* version of the matrix multiplication algorithm are as follows:

#p	min	avg	max
1	62.9	64.3	**66.7**
2	30.3	34.6	**40.8**
4	15.3	18.4	**20.4**
8	7.0	8.8	**10.3**
16	3.6	4.6	**5.2**
32	1.3	1.5	**2.2**
64	0.6	0.6	**0.6**
seq	58.8	70.3	**81.2**

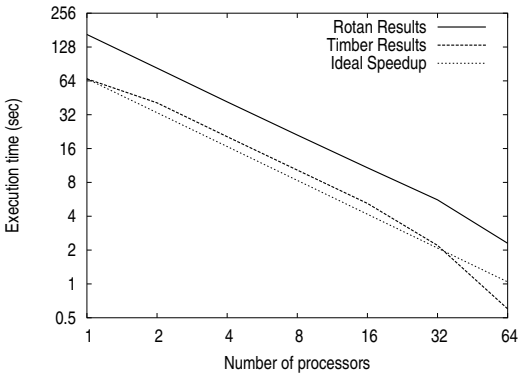


Figure 6.11: *Matrix multiplication, no communication; Rotan vs Timber.*

Analysis: The *Spar/Java* program about 2–3.8 times faster (in the truly parallel cases) than the equivalent *Vnus* program of Section 6.3.3. As there is no actual communication done, this can be entirely ascribed to the improved code-generation in the backend, specifically the faster array accesses.

In the next example, we take communication into account by changing the distribution of the arrays to match that of the *Vnus* program of Section 6.2.6:

```
static double A[*,*]
    <$on = (lambda (i j) P[(block i @bsize)])$> = new double[N,N];
static double B[*,*]
    <$on = (lambda (i j) P[(cyclic j)])$> = new double[N,N];
static double C[*,*]
    <$on = (lambda (i j) P[(block i @bsize)])$> = new double[N,N];
```

The results for this benchmark are:

#p	min	avg	max
1	64.7	67.5	**72.9**
2	87.7	91.3	**97.8**
4	72.1	81.0	**85.8**
8	45.6	46.6	**47.2**
16	23.0	23.4	**23.7**
32	7.9	8.2	**8.6**
64	4.3	4.3	**4.3**
seq	63.8	70.7	**81.3**

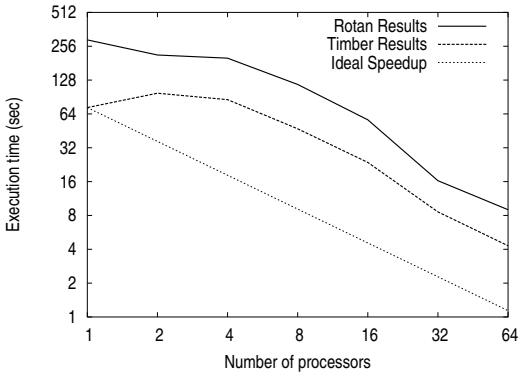


Figure 6.12: *Matrix multiplication with communication; Rotan vs Timber.*

Analysis: This curve has the same general shape as the version compiled by *Rotan*, and is 2–2.5 times faster in absolute sense in all truly parallel cases. Much of that, as we have seen, is explained by the better backend. The rest can be ascribed to the better rules for parallelism employed by the *Timber* compiler, e.g. the more aggressive optimising away of send-to-selfs at the source level, rather than leaving this for the run-time system to handle during run-time. This also explains why in the $p = 1$ case (when *all* communication will be a send-to-self) this *Spar/Java* program performs almost 4 times better than the *Vnus* program.

6.4.2 PGHPF

As a second touchstone, we used a commercial product: the High Performance Fortran compiler PGHPF from Portland Group Compiler Technology [Por01]. This is a full-fledged assembler-generating HPF compiler that has been commercially available since 1993. For our benchmarks, we used version 4.0-2, running on the DAS-2 supercomputer.

We again begin by investigating the case where no actual communication is done, corresponding to the *Vnus* program in Section 6.3.3.

The equivalent HPF matrix multiplication program is as follows:

```
    parameter(n = 1024)

    double precision, dimension(1:n, 1:n) :: A, B, C

!HPF$ DISTRIBUTE A (*,BLOCK)
!HPF$ DISTRIBUTE B (*, *)
!HPF$ DISTRIBUTE C (*,BLOCK)

    integer i,j,k

    A = 1.0d0
    B = 1.0d0
    C = 0.0d0

    do k=1,n
       forall(i=1:n, j=1:n)
          C(j,i) = C(j,i) + A(k,i) * B (j,k)
       end forall
    enddo
```

In Fortran, arrays are stored in column-wise order, as opposed to row-wise order for C or C++. In order to keep the comparisons fair, it is necessary to swap both the distribution specification (which now specifies `BLOCK` in the second dimension rather than in the first) and the array accesses (`(j,i)` instead of `(i,j)`). This does not change the actual algorithm, but ensures that the Fortran version will be able to make use of contiguous memory accesses and possible cache effects in exactly the same way as the *Vnus* and *Spar/Java* programs.

The timings for the program are as follows:

#p	min	avg	max
1	41.5	41.5	**41.5**
2	20.7	22.4	**25.1**
4	11.2	11.4	**11.6**
8	5.0	5.5	**5.9**
16	2.8	2.9	**3.0**
32	1.4	1.5	**1.5**
64	0.5	0.5	**0.5**
seq	39.8	44.6	**49.6**

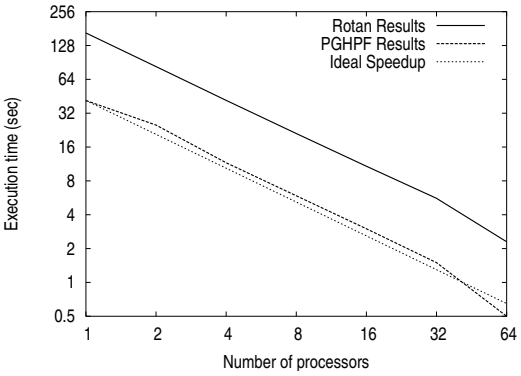


Figure 6.13: *Matrix multiplication with communication; Rotan vs PGHPF.*

Analysis: The Fortran program performs almost twice as well (40%) better than the *Spar/Java* program. This is not surprising, considering the fact that machine code is generated.

Next, we again introduce communication by changing the distribution on the arrays to match that of the *Vnus* program of Section 6.2.6:

```
!HPF$ DISTRIBUTE A (BLOCK,*)
```

```
!HPF$ DISTRIBUTE B (*, CYCLIC)
!HPF$ DISTRIBUTE C (BLOCK,*)
```

The results are shown in Figure 6.14:

#p	min	avg	max
1	49.6	49.8	**50.1**
2	25.2	25.2	**25.2**
4	12.7	12.9	**13.4**
8	6.7	7.1	**7.8**
16	4.9	5.3	**5.6**
32	7.8	8.1	**8.5**
64	23.3	23.3	**23.4**
seq	41.6	44.6	**49.5**

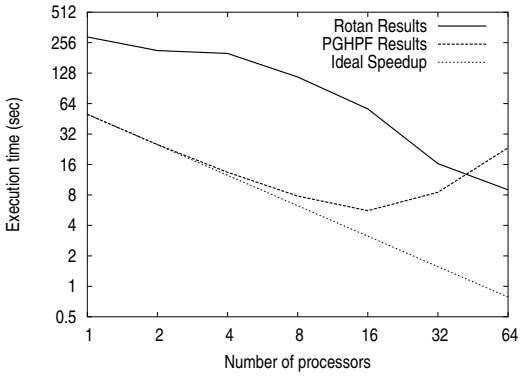


Figure 6.14: *Matrix multiplication; Fortran version with communication.*

Analysis: This is unexpected behaviour. The Fortran compiler somehow generates such inefficient communication templates, that the total execution time actually starts rising again for the very large values of p, although for $p < 32$ the program comfortably outperforms even the *Timber* compiler.

We suspect that the communication code generated by the PGHPF compiler is deliberately chosen such that it performs extremely well for a certain range of p, rather than attempt to be generically efficient, as is the case for *Rotan* and *Timber*, but lack of information about the internal workings of the PGHPF compiler makes it difficult to prove this theory, other than by looking at the results.

An interesting observation is that the effect seen here appears related to the use of the `forall` construct. If the `forall`s are replaced by conventional `do` loops (which the PGHPF compiler will still try to 'auto-parallelise'), the results are as shown in Figure 6.15.

Analysis: These results are uniformly worse than the previous version for $p < 16$, but uniformly better for $p > 32$, although the *Timber* compiler still performs better for $p = 64$ (and probably for larger p as well).

As investigating the peculiarities of the PGHPF compiler is not in itself the purpose of this chapter, we will not continue this discussion here. Instead we now turn to our second test case.

6.5 Red/black SOR

As a final comparison of the parallel programs generated by the *Rotan* system with the *Timber* and PGHPF compilers, we consider an implementation of red/black

#p	min	avg	max
1	88.5	88.5	**88.5**
2	88.4	88.6	**88.6**
4	44.2	44.2	**44.4**
8	11.5	11.5	**11.5**
16	6.7	7.2	**7.4**
32	5.1	5.6	**5.8**
64	5.6	5.6	**5.6**

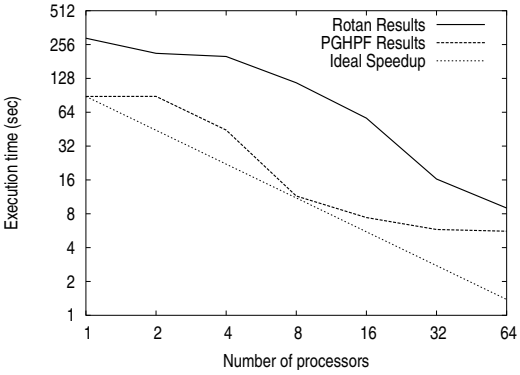


Figure 6.15: *Matrix multiplication; alternate Fortran version without* `forall`.

success overrelaxation (SOR).[1] This algorithm is more complex than matrix multiplication, and the resulting program is heavier on both computation and communication.

In its original form, the algorithm calculates a new version A' of a central matrix A in each iteration of its main loop. The program terminates when the maximum absolute difference between A' and A drops below a certain error threshold.

Expressed in *Spar/Java* (with some of the pragmas left out to enhance readability), the data structures and main loop look like this:

```
static final int N = 256;
static final int bsize =
               N / spar.lang.DataParallel.getNumberOfProcessors();

static double [*,*]
  <$on = (lambda (i j) P[(block i @bsize)])$> Grid = new double[N,N];
static double [*,*]
  <$on = (lambda (i j) P[(block i @bsize)])$> oldGrid = new double[N,N];
static double [*,*]
  <$on = (lambda (i j) P[(block i @bsize)])$> diff = new double[N,N];

// ...

  while (error > threshold)
  {
          // Store old values;
      foreach (i :- 0:N, j :- 0:N)
        oldGrid[i,j] = Grid[i,j];


          // Compute even points
      foreach (k :- 1:N-2:2, m :- 1:N-2:2)
         Grid[k,m] =
           (Grid[k+1,m] + Grid[k-1,m] + Grid[k,m+1] + Grid[k,m-1])/4.0;

      foreach (k :- 2:N-1:2, m :- 2:N-1:2)
```

[1]The SOR implementations discussed in this section are based on an original HPF version by S. Elmohamed [Elm96].

```
        Grid[k,m] =
          (Grid[k+1,m] + Grid[k-1,m] + Grid[k,m+1] + Grid[k,m-1])/4.0;


         // Compute odd points
      foreach (k :- 2:N-1:2, m :- 1:N-2:2)
        Grid[k,m] =
          (Grid[k+1,m] + Grid[k-1,m] + Grid[k,m+1] + Grid[k,m-1])/4.0;

      foreach (k :- 1:N-2:2, m :- 2:N-1:2)
        Grid[k,m] =
          (Grid[k+1,m] + Grid[k-1,m] + Grid[k,m+1] + Grid[k,m-1])/4.0;


        // Compute error
      foreach (i :- 0:N, j :- 0:N)
        diff[i,j] = abs(Grid[i,j] - oldGrid[i,j]);

      error = 0.0d;
      foreach (i :- 0:N, j :- 0:N)
        error = max(error, diff[i,j]);


        // Compute and store new grid value
      foreach (i :- 0:N, j :- 0:N)
        Grid[i,j] = oldGrid[i,j] + omega * (Grid[i,j] - oldGrid[i,j]);
  }
```

Since computing the error value involves a reduction operation which *Rotan* has no support for, the algorithm was adapted to loop a finite but large number of times (12,711), which is sufficient to calculate *Grid* within an error margin of 0.01 for a matrix of 256 by 256 elements.

The results for each of the three systems are given in Figures 6.16, 6.17 and 6.18, while Figure 6.19 provides an aggregate view of the same results.

#p	min	avg	max
1	854.0	857.4	**863.4**
2	431.0	454.7	**499.7**
4	209.4	216.0	**219.6**
8	79.2	83.4	**86.1**
16	33.2	33.6	**34.1**
32	17.2	17.7	**18.4**
64	15.1	15.5	**16.5**
seq	230.5	248.3	**275.5**

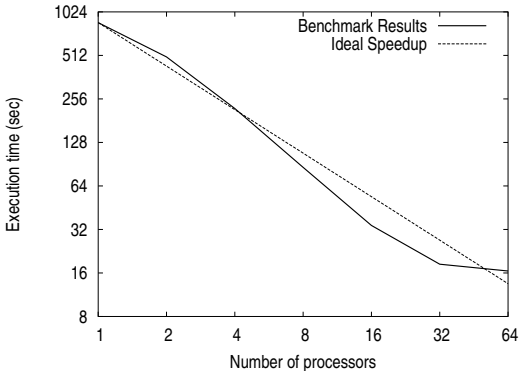


Figure 6.16: *SOR;Vnus version, compiled by Rotan.*

Analysis: These figures illustrate that the benchmarks obtained for the simpler matrix multiplication program already offer a good indication as to how a more complex program will behave. All three compilers create code that show speedup, although the absolute numbers vary.

#p	min	avg	max
1	218.9	219.8	**220.6**
2	95.3	102.3	**107.5**
4	23.3	26.4	**32.5**
8	10.2	10.2	**10.3**
16	6.5	6.5	**6.5**
32	5.0	5.0	**5.0**
64	5.0	5.0	**5.0**
seq	199.4	217.4	**245.1**

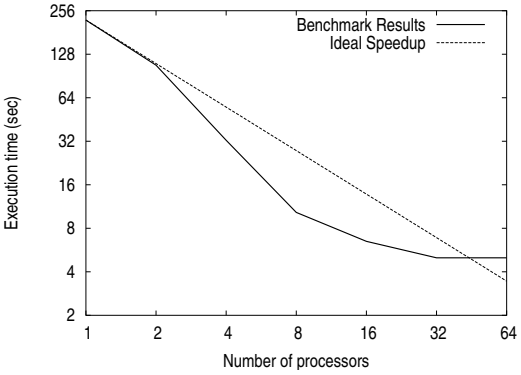


Figure 6.17: *SOR;Spar/Java version, compiled by Timber.*

#p	min	avg	max
1	223.1	238.8	**246.7**
2	107.4	107.8	**108.3**
4	44.5	44.6	**44.7**
8	14.9	15.9	**16.9**
16	10.0	10.1	**10.2**
32	8.7	8.9	**9.1**
64	9.0	9.1	**9.1**
seq	210.0	218.4	**231.1**

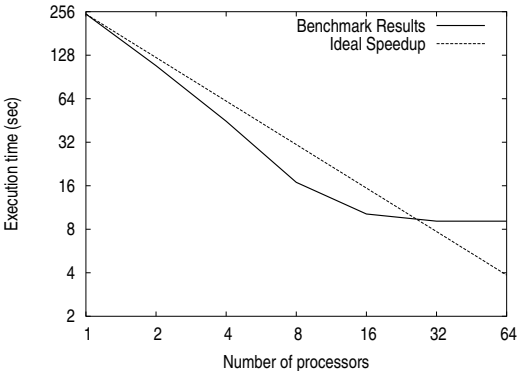


Figure 6.18: *SOR;HPF version, compiled by PGHPF.*

In the sequential case, the *Timber*- and PGHPF-generated programs are, both at 245 seconds, only about 10% faster than the *Rotan* code.

For the truly parallel cases, however, the *Timber* results are 3.3 ($p = 64$) to as much as 8.4 ($p = 8$) times faster than *Rotan*. Partially we have already seen that this can be explained by the better backend and decreased overhead. Another factor is also that *Spar/Java* has a special pragma for identifying stencil operations such as found in the SOR main loop to the compiler. If this pragma is *not* used, *Timber* is only 7.6 times faster for $p = 8$.

The Fortran program (here again executed with a column-wise distribution and traversal of the matrix index space) shows the same overall speedup curve, but does not parallelise as well as the *Timber* version. For $p = 2$, the two versions perform the same, but for each additional processor the Fortran version starts lagging behind a bit more, until by $p = 64$ it is a full 45% slower.

These results support our earlier conclusion that the PGHPF compiler, while

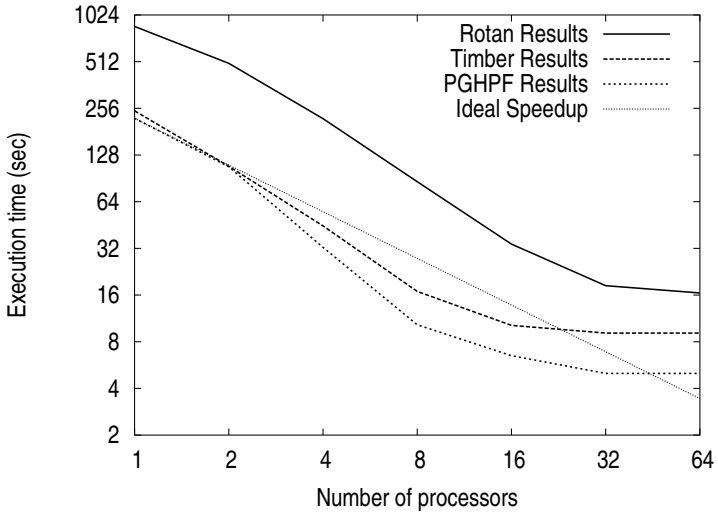


Figure 6.19: *SOR;Rotan vs Timber vs PGHPF.*

better at generating efficient computation, simply does not appear to use communication algorithms that are as scalable as the ones used by *Rotan* and *Timber*.

6.6 Conclusion

The concept of rule optimising engines is proving itself. Rules can have a large impact on the shape and position of a benchmark curve (and are therefore not just trivial toy examples), and being able to turn them off and on facilitates experimenting and analysis.

The *Rotan Vnus* compiler is, in the worst cases about 7.5 times, and in the best cases about 2 times slower than the less experimental compilers. This can plausibly be explained by the improved backends and run-time systems for those other compilers, and in the *Rotan Vnus* compiler's lack of certain specialised optimisation engines that could improve performance by getting rid of useless send-to-selfs, or by performing tiling-specific communication tweaks.

In the last chapter of this thesis, we will use these performance results as part of the overall evaluation of the *Rotan* transformation system.

Chapter 7

Evaluation

The previous chapters of this thesis have shown that the concept of an instantiable, rule-based compiler is a valid and useful one in order to manage the complexity involved in the translation and optimisation of data-parallel programs.

In Chapter 6 we have evaluated the *Rotan* system in terms of the efficiency of the code it produces. In this chapter, we will evaluate the *Rotan* system (and specifically its core component, the *Rule Language*) against the original design criteria for the system, and examine how well the prototype implementation has stood up in different areas of use. Our examination will focus on the following areas:

Applicability to different domains. The main test case we have presented in Chapter 5 involved the implementation of transformations for the *Vnus* domain, but how well-suited is *Rotan* to handling other domains and languages?

Applicability of embedded code. The possibility to escape to embedded code is one of the strengths of the *Rule Language*, but at the same time something a compiler writer does not want to have to resort to it too much. How often, and to what extent has embedded code actually been necessary in the *Rotan Vnus* compiler?

Expressive power of the *Rule Language*. Related to the previous points is the question of how powerful the *Rule Language* is (or where its limits lie), and to what extent it allows the programmer to specify arbitrary transformations with ease.

Readability of the *Rule Language*. Does the *Rule Language* manage to avoid being a 'write-only' language?

Portability of the implementation. Is the *Rotan* system itself well-written, and in particular: portable to other systems and platforms?

Performance of the system in time and space. The efficiency of the generated code is of paramount importance in compiling for parallelism, but even though we explicitly prefer compile-time performance penalties over run-time performance penalties, the compilation system must perform within certain constraints in order to be a realistic alternative to a conventional black-box compiler.

Development environment and support tools. The *Rule Language* and the transformation engine are not the totality of the *Rotan* system. How well is rule programming supported by other components of *Rotan*?

For each of these areas, we discuss the original design considerations and decisions, and we evaluate the actual implementation as it currently exists. Subsequently, we offer suggestions for future changes and improvements, both short-term and long-term.

These evaluations of *Rotan* against its own design criteria are useful, but also a bit more removed from the reality that prompted those design criteria in the first place. In order to further ground our evaluation of the *Rotan* system in a real-world context, we will also compare *Rotan* against the *Timber* compilation system for data-parallel programming languages developed at the Parallel and Distributed Systems group at the Delft University of Technology.

7.1 Applicability to Different Domains

7.1.1 The Design

During the course of the research described in this thesis, the *Rule Language* has been applied to a number of different domains.

The *Rotan* system has served as both backend and frontend in addition to being used for parallelisation and optimisation of the intermediate language *Vnus*. As a frontend, *Rotan* converted programs written in the high-level parallel language *Booster* into the intermediate format *V-cal* (a precursor of *Vnus*). As a backend, it mapped *V-cal* directly to a C++-like target domain.

Proof-of-concept instantiations of the *Rule Language* were made for the *Vista* language [Jon91], a transaction-based programming language developed in the *ParTool* project [Ste92], and for the functional, intermediate language *F-Code* [Muc93]. These instantiations included the definition of a domain file, and the creation of a small number of test rules. The experience with these experiments was positive, and did not give rise to doubts about the system's ability to describe transformations on parse trees in these domains.

7.1.2 The Implementation

Each domain is implemented as a set of generated C++ classes (one class for each node type in the domain), originating from a single *domain description file*. This lower-level interface has in fact seen several iterations and major rewrites. The first implementation used *lex* and *yacc* to parse a domain description file, and the custom tool *libgen* to generate a set of macro-filled source files from the results, which finally yielded the C++ source code when put through the C preprocessor.

This entire process was painfully slow (because the preprocessor had to be called for every file), error-prone, and yielded intermediate files that became progressively larger, more unreadable and more difficult to debug with each further step. The system itself was so complex as to be unmaintainable.

Eventually, this approach was abandoned altogether, and the decision was made to instead use the template managing system *Tm* [Ree92], which is specifically suited to the creation of template-based code. This change has been successful. Domains are now described in *Tm* data structure files, and the generation of the C++ classes is has become fast, stable, and manageable. Generating the code for the entire C++ class hierarchy (205 classes) takes only seconds on a standard issue 350 MHz desktop Pentium-PC, as opposed to the minutes it took when the C preprocessor was used.

7.1.3 Evaluation and Future Suggestions

The capability to use domains as a means of instantiating the *Rule Language* for a specific problem space has been one of the most successful concepts in the systems. There are no known problems with the current implementation, nor with the current language constructs.

One possible direction of future research is the addition of a type system to the *Rule Language* with respect to entire domains (not just with respect to nodes in the domain). It would then become possible to warn the programmer if rules are written that go "in the wrong direction", i.e. from target domain to source domain, or to warn if at the end of a run parts of the input tree are still untransformed. This can already be largely implemented with the type system for nodes currently in place, but additional support could be given to the user if the *Rule Language* would be extended in this way to recognise and be able to handle multiple domains in a single instantiation.

7.2 Applicability of Embedded Code

7.2.1 The Design

The *Rule Language* was designed with the goal of making basic transformations on a domain tree easy by providing constructs in the language, and complex transformations possible without turning the language unwieldy.

In order to accomplish the latter, hooks were provided in the *Rule Language* for attaching straight C++ code to individual rules, both in the match patterns and in the action patterns. Embedded code provides full access to the domain tree, and therefore adds unlimited potential to the rule. This comes at the expense of type safety (the system has only very limited knowledge about what is specified in an embedded code block) and at the risk of interfering with the proper application of the rule itself (by bringing the domain tree into an inconsistent or invalid state).

In the *Rotan Vnus* compiler, 37% of the rules contain embedded code. As embedded code is intended as an if-all-else-fails escape option, it is important to investigate these instances. Excessive use of embedded code indicates a lack of expressive power in the *Rule Language* proper.

A closer examination of the rules in the *Rotan Vnus* compiler, yields the following break-down with respect to the different types of problem solved by use of embedded code[1]:

1. 43%: Creating a copy of a part of the matched program tree (e.g. cloning operations used when shared subtrees are not desirable). This is a one-statement operation.

2. 42%: Constructing a new part of the program tree from scratch (e.g. creating unique new identifiers for newly introduced variables). This too is a one-statement operation.

3. 15%: Other. On average, these embedded code blocks are only about four lines long.

From this list we can see that the vast majority (85%) of embedded code used in the *Vnus* compiler serves to implement two very primitive operations, rather than to add truly unique functionality to a rule. A future version of the *Rule Language* might be expanded to encompass these primitives (offering the user e.g. a built-in cloning operator).

Some changes have already occasionally been made over the course of the *Rule Language*'s lifetime. For example, the ability to compare matched nodes of the String and Integer types against literal constants directly from within the *Rule Language* itself, was at one point incorporated into the rule language when previously it had to be performed in embedded code instead. This allowed more than a third of the then existing embedded code to be replaced by direct comparison in the where-clauses.

Embedded code of the cloning type indicates a specific issue in the *Rule Language* design: rule variables are implemented as pointers, and in the action part of a rule it was decided to give the use of a rule variable the semantics of a pointer copy rather than that of a deep copy of the structure the pointer is pointing to. Both semantics have advantages and disadvantages, depending on the type of

[1]The percentages are based on lines of code, and are very approximate.

rule. In retrospect it might have been a better idea to incorporate into the *Rule Language* the possibility to offer the user both types of semantics.

Embedded code of node construction type raises another issue: although string literals are built into the *Rule Language*, we find that escape to embedded code is still necessary for the purpose of creating *unique* strings, i.e. a sequence of identifiers of the form **`var1`**, **`var2`**, **`var3`**, etc, such as is used during the insertion of temporaries. The number of necessary variables is not known during compile time, because it can for instance depend on the number of successful matches for the rule. In the current *Vnus* compiler a custom-written *idGenerator* class is used to dynamically create unique strings from within embedded code. This is also a functionality that could be lifted into the *Rule Language* itself, although here the target is much less generic than in the cloning case. Care should be taken not to overload the language with too many 'handy' constructs.

The third type of embedded code (the 'other' category) is a catchall category that contains some odds and ends, such as code for a specific rule that needed to perform a simultaneous iteration and comparison over two lists (which is not supported by any *Rule Language* construct), and code that tests for a more complex relationship between rule variables than equality.

7.2.2 The Implementation

The current implementation for embedded C++ support is straightforward. All embedded code in a rule is first run through a filter that changes the references to rule variables to references to the corresponding C++ pointers. The filtered C++ code is then copied to a separate source file, compiled, and linked back into the *Rotan* system.

The rule source files are generated by the rule system itself from hard-coded C++ code. This makes them inflexible, difficult to maintain, and causes unnecessary recompilations. Since the rule source files all have the same general structure, a more efficient implementation would use a more generic template-supported approach, possibly using the template manager *Tm*.

7.2.3 Evaluation and Future Suggestions

Embedded code has proven valuable for increasing flexibility, but also for working around problem spots or missing functionality in the *Rule Language* design. Most of these problems turn out to be minor, and can be addressed in a future version of the *Rule Language*.

The implementation of the handling of embedded code consists of code that is complex and error-prone. While a future version of the rule system would be improved by reimplementing the embedded code support, the interface to the *Rule Language* has proven adequate and can remain as it is, although more knowledge and compile-time checking of embedded code from within the *Rotan* system itself would be welcome.

7.3 Expressive Power of the *Rule Language*

7.3.1 The Design

The *Rule Language* has been designed primarily as a practical language, intended to solve existing problems fast and quickly. As such, the language design was steered to a great extent by practical concerns, leading to such properties as the embedded code hooks, the domain-parameterisability, and the direct support (through lists, expressions and nodes, see Chapter 4) for domains representing programming languages.

Although never the main focus, attention was also given to other, more general features of language-design, such as type-safety, abstraction, and code reusability.

The *Rule Language* that has emerged from this result-oriented approach can, in some of its aspects, be compared to the C language: powerful and high-level enough to be able to accomplish virtually anything in a structured, readable way; but also with some low-level constructs and idiosyncrasies that can take time to master, and that do not always lead to as elegant a program as one might like.

7.3.2 The Implementation

In this section we enumerate what have emerged as noticeable design issues with the implementation of the *Rule Language*. All these problems can be (and have been) worked around.

- *Flow control constructs.* Rules are like single statements in an imperative language. Each rule causes a change in the program state, and although rules can be applied iteratively, this iteration is coarse-grained and cannot be influenced (let alone defined) by the programmer — it is all or nothing. Basically every rule is an atomic event. Drivers and Engines are sequential orderings of rules — compound statements, to continue the imperative language analogy. The *Rule Language* would be improved by better control flow for rules. It should be possible to try rules conditionally using if-then-else constructs. It should be possible to gain finer control of iterative rule application using while, repeat, or for-constructs. It should be possible to group sets of rules into true subroutines, and be able to apply those, collectively. Finally, it might be necessary to allow the programmer a means to influence or define the traversal and application strategy followed for specific rules.

- *Re-entrant rules.* A rule will always work on the *entire* program tree. Even in those cases where this appears to be not the case, the entire program tree is implicitly used as the source for the rule. This makes it impossible to apply rules from within other rules (another desirable aspect of *Rule Language* flow control), and hence causes code duplication that might otherwise be avoided.

- *Rule variable arguments.* Currently, rule variables are entirely local to a rule. In fact, under certain circumstances rule variables are local to only a *part* of a rule. Once better control flow for rules becomes part of the *Rule Language*, it will also become desirable to be able to refer to parts of the program tree *outside* of individual rules, and to pass parts of the program tree as arguments to rule routines or even individual rules.

- *Lists, Expressions, Nodes.* Lists, expressions, and nodes form the key structural components of the *Rule Language*. These constructs have become closely entwined with their counterparts in the underlying implementation. These same structures are used not only in the program trees, but also in the implementation of the *Rotan* system itself. Our experiences with writing rules have shown that the decision to incorporate these structural entities into the *Rule Language* was in itself a good one, but the implementation of domains and the system needs to be kept more separate. The use of typed lists and typed expressions should be investigated in particular, as well as better ways of manipulating individual list elements and sublists. The same holds for nodes that allow subscripting.

- *Parameterised creation of program tree fragments.* One of the main reasons for escaping to embedded code (see the previous section) is the fact that trees created as the result of the action part of a rule cannot pass parameters to the underlying constructors: they are created using only the default, parameter-less constructors in the corresponding C++ class. Any members taking non-default values must be explicitly constructed in embedded code. The *Rule Language* can offer much better support for such actions.

7.3.3 Evaluation and Future Suggestions

Most of the implementation changes discussed above will necessitate a non-trivial redesign of the core transformation engine, although the essence of the rule system would not be changed: individual rules would still match and fire in the same way that they do today.

7.4 Readability of the *Rule Language*

7.4.1 The Design

In general, rules are more difficult to read than to write. Small rules can be very elegant (particularly if no embedded code is involved), but the more complex rules quickly become cluttered and hard to follow, even with careful formatting.

The very fact that the *Rule Language* is parameterisable adds considerably to its complexity. Even though the number of constructs and keywords provided by the language proper is quite small, each domain will generally contribute a

complete hierarchy of node types. This means that a rule can be difficult to understand if the reader has no knowledge of the domain. Even if that knowledge is present it may be difficult to have all that knowledge at immediate recall when reading a rule. For example, someone attempting to understand a *Vnus* transformation rule must not only understand the *Rule Language* constructs used, but must also understand the entire *Vnus* language and have a good idea of what a *Vnus* program tree would look like.

A second factor is that a transformation rule is an attempt to write down in a linear, one-dimensional sequence, something that is inherently two-dimensional (tree structures), and has an associated time-axis as well. The 'match' part of the rule corresponds to an initial program state, the 'action' part of the rule corresponds to the program state resulting from the firing of the rule. This time axis can be adequately represented in the sequential form used by the *Rule Language*, since it is a universally established programming convention that, when reading top-to-bottom, subsequent units of code represent subsequent events in time. Unfortunately, the two-dimensionality of each rule-part is not so easily mapped to a linear sequence. The program tree represented by each part is a two-dimensional graph, with potentially unlimited branching potential, and potentially unlimited depth. *Rule Language* programmers can attempt to map this to a readable text by using vertical indentation for representing branching, and horizontal indentation for representing depth, leading to a 'toppled' representation of the program tree.

This approach leads to obvious problems: deep rules will quickly extend too far to the right, rules that branch too widely will become too large, and in general the structure of the tree will quickly become difficult to follow in the mass of text.

7.4.2 The Implementation

The current implementation of the *Rule Language* is not affected by the above considerations, since the mapping, however difficult to read for humans, is unambiguously parseable. For the computer it does not matter how complex the rule is — it will always be possible to assemble a rule-tree for comparison with the actual program tree.

7.4.3 Evaluation and Future Suggestions

The *Rule Language* works as a language, and as an interface to the transformation system, but can be difficult to use for humans, as soon as the rules approach a certain complexity, merely because of readability problems.

For future research we propose to view the *Rule Language*, similar to *Vnus*, primarily as an intermediate language for machine use. A higher-level representation is needed to further hide the structural complexity of the *Rule Language* from the programmer. Considering the above-mentioned aspects, a possible avenue to explore for a more user-friendly *Rule Language* can be that of a graphical

language.

If we have a language that works on trees and tree fragments, the programmer should be able to actually *work* with those trees and tree fragments. A combination of graphical editor and folding editor could be written, where entire branches of a tree can be collapsed or expanded, and conditions or code or actions attached to individual nodes. The resulting graphical program would be represented as a conventional *Rule Language* program, and should purely be seen as an interface on top of the language.

It would also be advisable to base such a language from scratch on a more sound theoretical basis than has been the case for the more ad-hoc *Rule Language*.

7.5 Portability of the Implementation

7.5.1 The Design

Since the *Rule Language* is mostly self-contained, there are few portability issues at the language level. The only direct interface to the outside world is through the embedded code. This does mean that the *Rule Language* is dependent on the existence of C++ to be usable on a certain platform, and may be affected by differences at that level.

However, the connection between the *Rule Language* and the embedded code has purposely been kept weak, so that should the need arise it would be trivial to allow embedded code in other languages as well. This would of course have severe implications for the implementation of the system, and all existing rules using embedded code would have to be rewritten, but the language as such would not be affected.

7.5.2 The Implementation

The *Rotan* system is implemented in C++, and only uses auxiliary tools that themselves are written in C or C++. The resulting portability of the entire system has proved itself useful many times. The original implementation of the rule system, which was on Sun 3 and Apollo workstations, has since then been successfully, and with a minimum of effort, been ported to different hardware platforms (HPs, SPARCstations, Intel processors) running different operating systems (HPUX, SunOS, Solaris, and various flavours of Linux).

For reasons of portability it was also decided at an early stage not to use any third-party libraries or classes, but develop all support classes in-house. In retrospect, this was an unfortunate decision. Support classes are not trivial to write, and have, rather then helping us remain portable, caused more portability problems by themselves than anything else in the system. Over the last few years the C++ language has evolved considerably as well, and for instance the emergence of the Standard C++ Template Library eliminates the very need for most of these support classes (strings, lists, arrays, etc.).

So although we have taken care to ensure that the portability of the system has never been compromised, it should be noticed that most of the code has become dated in the sense that we are not making use of the more recent developments in both the language and the language support.

7.5.3 Evaluation and Future Suggestions

On the whole, using C++ as an implementation vehicle has been successful. Although the core of the transformation system is portable and well-written, the support classes are still riddled with legacy code, which has had detrimental effects on the quality of the rest of the system as well.

A new version of the system would be a solution, and C++ would still be an acceptable implementation language. The implementation of the core transformation engine could for the larger part remain identical, but a rewrite from the ground up, incorporating the latest C++ language constructs and support libraries, should result in increased maintainability, as well as in substantial code size reduction.

7.6 Performance of the System in Time and Space

7.6.1 The Design

Performance of the *Rotan* system itself (as opposed to the performance of the programs created in it) has only been a secondary design goal, both in terms of execution speed and in terms of memory usage. There have been a few exceptions:

Compiled rules. During the early phase of the project, a version of the rule system was created in which rules were compiled out into direct C++ code, rather than interpreted. Eventually, the problems associated with maintaining two nearly identical versions of the system became too bothersome, while the speed gains, if any, were not noticeable at all to the user.

Partial treewalks. We also experimented with an addition to the *Rule Language*, where specific rules were given more information about the types of nodes they were applicable to, thus making it possible for the system to skip certain parts of the program tree during attempts to match or fire a rule. This experiment was successful in concept, but the implementation was not considered stable or elegant enough to be retained or developed further afterwards.

Reuse semantics for rule variables. As was mentioned earlier, the use of rule variables in action patterns is implemented as a pointer copy, not as a tree copy. This decision was at least partly made because of memory considerations.

7.6.2 The Implementation

As mentioned, the *Rule Language* is an interpreted language, and will therefore never be a speed demon. In practice, application of even the largest set of transformation rules to our largest program has generally been so fast (on a 350 MHz Pentium-PC programs such as our matrix multiplication test case compile in a matter of seconds, and even the largest test program takes no longer than a minute) that no need for further development in this area was perceived. Similarly, the workstations on which development has taken place were also well-equipped in memory, and no problems in that area were ever encountered, despite the fact that the implementation of the rule system allocates a large number of class instances — and hardly ever releases any of them before the program ends. For truly complex programs, scalability and performance may become more of an issue, however.

7.6.3 Evaluation and Future Suggestions

Performance and memory usage are currently not a problem, but the boundaries are being reached, and a future implementation of the *Rule Language* should take care to get the memory allocation right from the start, if only because this will lead to greater program stability. If performance ever becomes a problem, then both performance enhancements, compilable rules and pruned treewalks, could be reimplemented.

A different approach would be to take advantage of parallelism and actually execute rules in parallel (see for instance the *Parlanse* language [Bax97]). The *Rule Language* would have to be extended for this to work — whether or not rules can be executed in parallel is difficult to find out by analysis alone, considering the effects of embedded code. Presumably new language constructs (for instance in the form of pragmas and annotations) could be used to support this.

7.7 Development Environment and Support Tools

7.7.1 The Design

By far the least developed area of the current *Rotan* system is the support environment used for interactively writing, testing, and debugging rules. The current environment, known as *rcc*, consists of a simple command line environment with support for actions such as loading and displaying programs; and loading, displaying, and applying rules. In combination with a decent editor such as emacs this was sufficient to develop the rules used in this thesis, but this environment would not be of much use to a novice rule programmer.

7.7.2 The Implementation

The current implementation of *rcc*, as the rest of the rule system, is in portable C++, and can be used a basis for a new version. The ultimate solution would be to create a proper graphically-oriented environment (mouse, windows, etc.), but the complete list of requirements for such an environment is beyond the scope of this thesis.

Failing the availability of the resources for such an extensive redesign, a future version of the rule system could retain the command line interface, but should at the very least incorporate support for proper command line history, for the handling of command line editing keys, for scrolling through files larger than the current display, and for batch processing of rules (in effect implementing a command-line compiler version next to the interactive version).

Support for different domains is adequately present, but the feedback towards the user can still be much improved.

7.7.3 Evaluation and Future Suggestions

The current environment is more than adequate for a prototype, experimental version.

7.8 Comparison with *Timber*

The *Timber* compiler [Ree03a; Ree03b], like the *Rotan* system, was developed by the Parallel and Distributed Systems group at the Delft University of Technology. There are both similarities and differences between the two systems, which makes it interesting to compare and contrast them.

One difference is that *Timber* has seen more development effort and iterations over the years than was possible for *Rotan*. Where *Rotan* is very much still an extended prototype, *Timber* is a much more mature system, that can, in the area of optimisation, hold its own in comparison against commercial systems.

Another major difference is that *Timber* is a special-purpose compiler, dedicated to the compilation of *Spar/Java* and the processing of *Vnus* program trees. In contrast, *Rotan* is a more generic system, with the whole domain plug-in structure designed to easily accommodate other languages and formats. The instantiated *Rotan* compiler for *Vnus* described in Chapter 5, however, basically covers a sizable subset of the compilation and optimisation trail that *Timber* implements, and is therefore quite comparable, as we have already seen in Chapter 6.

Another similarity that has already been mentioned is that both *Timber* and *Rotan* (instantiated for *Vnus*) use the same frontend, backend, and run-time system libraries (albeit frozen older versions in the case of *Rotan*), and that both systems make extensive use of the *Tm* Template Manager [Ree03b] to describe data structures and to generate code. Both systems also use rewrite rules to

implement the actual program tree transformation, but this is where the two systems are both very different *and* very similar.

- The similarity lies in the fact that during the development of the two sets of rules there has been a large amount of mutual design and code sharing. The *Rotan* project predates the *Timber* project considerably, and the first set of *Timber* rules were based on the already existing *Rotan* rules. Later on, after progress on *Timber* picked up speed and surpassed the by then stabilised *Rotan* system, some (but by no means all) of the newer rules and algorithms implemented for *Timber* were ported to the *Rotan* ruleset.

- The difference lies in the implementation of the rules themselves. *Rotan* uses the *Rule Language*, *Timber* chose a more low-level approach, in which *Tm* templates are used by the compiler builder to generate treewalkers for analysis and transformation purposes.

The *Tm* treewalkers offer an interesting way to process a parse tree node by node. The action to be performed on each type of node is explicitly specified by the programmer, but the code necessary to traverse the tree and ensure that all instances of the target nodes are visited (and the action code subsequently applied) is generated. The user specifies a list of all the node types that are to be visited, the action functions for all these node types, and some macros for generating signatures and invocations of the walker functions. From this information *Tm* computes the appropriate walkers. By letting the user specify the signature and the action of the walker functions, arbitrary information can be passed into or accumulated during the tree walk. Tree walkers are similar in concept to the Visitor pattern in object-oriented programming.

We can compare the *Rotan* and *Timber* approaches to compilation based on the aspects used earlier in this chapter to evaluate *Rotan* itself in a more absolute sense.

Applicability to different domains. *Rotan* was designed and has been proven to easily accommodate different languages/domains. Although the *Timber* infrastructure is well-designed and modular, and much of it has been re-used in other projects, *Timber* was never intended to be a general-purpose transformation system.

Applicability of embedded code. *Timber* stays much closer to the C++ level than *Rotan* does. In *Rotan*, rules are composed in the *Rule Language*, and embedded code is used when necessary, but is basically to be avoided as much as possible. In *Timber*, the *Tm* templates are themselves C++ code expanded with special directives in *Tm*'s template language. The generation of infrastructural code is handled (and hidden from sight) as much as possible, but all the actual visitor code is expressed directly in C++. The *Timber* compiler uses not only the custom treewalkers, but also various

standard templates (that generate C code) for data structure manipulation and administration. In total, the *Timber* system consists of about 26% handwritten code, 39% code generated by *Tm*'s standard templates, 34% code generated by the custom *Tm* treewalkers and analysers, and 1% remaining code generated by *yacc*.

Expressive power. Both systems are equally powerful in the sense that both can use handwritten C++ to the full extent necessary. In terms of the abstractions offered, however, *Timber* offers more complex infrastructural functionality than *Rotan*. In the *Rule Language* escaping to C++ is the only solution if something is not provided by the *Rule Language* itself, and as there is no further support beyond the escape to lower-level code itself, the compiler builder is left with nothing more than a data structure to work with. The closer link of the *Tm* templates to the C++ code, however, means that while support for things such as tree-traversal is not as hidden away from sight as in *Rotan*, there is also more support available for creating new functionality more or less from scratch. In essence, *Timber*'s expressive power takes place at a lower level of abstraction than *Rotan*'s, making development easier for the programmer who needs to work at that level, whereas *Rotan*'s expressive power lies in the higher level.

Readability. Readability will always be a partially subjective evaluation, but as can be deduced from the previous point, the higher abstraction level of the *Rule Language* does make *Rotan* rules easier to read than *Timber* rules, in the same way that C programs are easier to comprehend than assembler programs.

Portability of the implementation. Both systems, including their supporting environments, are written using standard, portable ANSI C and C++. There are no relevant differences here.

Performance of the system in time and space. The *Rotan* system is a research prototype, and the performance of the system itself was always a secondary concern. *Timber*, while also used mostly for research purposes, is an actual released system, that has been the recipient of a sustained nearly full-time development effort spanning more than seven years time. It is no wonder that the performance of *Timber*, specifically the performance of the application of rules, is significantly better than that of *Rotan*.

Development environment and support tools. Although the *Rotan* development environment is fairly rudimentary, it is an improvement over the *Timber* situation, where there really no special support for writing or debugging rules, beyond that what the C++ and *Tm* environments offer — which is so generic in nature that it will not specifically help anybody trying to develop transformations at that same level of abstraction. *Timber* does,

however, have support for establishing and performing regression tests that is lacking in *Rotan*.

Both *Rotan* and *Timber* clearly illustrate that rule-based compilation systems for data-parallel languages are a superior alternative to systems that confine themselves to the implementation language's level of abstraction. Whether the rule-based compilation is achieved through something as high-level as a separate rule language, or something a bit less abstract such as template-based meta-programming, is secondary to this primary observation.

Timber's template-based lower-level support makes it more generically powerful, and the system is certainly more polished in its implementation than *Rotan*. However, daily practice with writing, maintaining, and understanding rules for both systems has shown that *Rotan*'s support for higher levels of abstraction (via the domains and the *Rule Language*) is a very strong point in its favour, leading to quicker development, better readability, and easier maintainability and expandability of rules than is the case for equivalent *Timber* rules.

In the next, final chapter of this thesis we will summarise and conclude our evaluation of *Rotan* and the research presented so far.

Chapter 8

Conclusion

8.1 Discussion

It has been argued (e.g. in [Paa92]) that a valid approach towards solving the 'parallel programming problem' as described in Chapter 1 lies in the creation of high-level parallel programming languages that abstract away from the underlying complexity of parallel hardware and programming models. A direct corollary of this is that the compiler technology used to translate such languages also has to evolve in order to meet the increased demands for extracting efficient parallelism from these higher-level specifications.

The work presented in this thesis shows that a programmable, rule-based compiler based on a high-level transformation system is in turn a fruitful way of implementing this evolution, at least for the field of data parallelism (Chapter 2). By having the compilation system itself make a leap towards a higher abstraction level, so that various complex transformations can be expressed in a way that is lucid and modular, we can substantially increase a compiler's own maintainability, efficiency and power. The *Rotan* system, with its associated *Rule Language* (Chapters 3 and 4) forms the prototype system serving as an implementation vehicle for our research into creating an optimising compiler for the *Vnus* intermediate language (Chapter 5).

The question whether the programmable compiler's own increased abstraction level sacrifices the efficiency of the code it generates is an important one. The performance benchmarks and comparisons with other compilation systems presented in Chapter 6 suggest that it does, but only to a quite acceptable extent. The parallel code generated by *Rotan* is about 2–8 times slower than that generated by more professional systems, but these slowdowns can generally be traced and attributed to specific causes (such as e.g. a limited backend implementation, or an as-yet missing optimisation) that do not imply inherent deficiencies of the trans-

formation system. In contrast, the shortened development time for new rules, their general expressive power, and their maintainability are strong advantages to frameworks such as *Rotan* (Chapter 7).

8.2 Future Work

There are many interesting directions in which future research on programmable compilers can be taken.

From a practical viewpoint, the current *Rotan* system can be expanded to remove a number of its limitations and improve its general capabilities and usefulness. The *Rule Language* can be extended with support for constructs such as re-entrant rules and flow-control (see also Chapter 7). It would also be valuable to allow programmers the possibility to explicitly specify custom tree-traversal strategies, rather than be limited to the three options currently available. Another interesting option would be to offer a more direct programming interface (expressed in high-level *Rule Language* constructs) to an underlying template-based code-generating mechanism (à la *Tm*). In effect this could unlock and incorporate an intermediate level of rule-programming that lies between the *Rule Language* at the top and the completely unchecked embedded code at the bottom.

Only lightly touched upon in this thesis is the fact that the intermediate language *Vnus* is formally defined with a corresponding calculus for reasoning about *Vnus* programs ([Dec98]). It is a logical extension of this theoretical basis that we might also wish to reason about transformations applied to a *Vnus* tree, and for instance prove the semantics-preserving aspect of a given transformation. This can lead not only to an additional level of confidence in *Rotan*/*Vnus*-based compilation systems beyond that offered by traditional systems based on a more ad hoc intermediate format, but it also opens the way for allowing *Rotan* support based on these more theoretical aspects of *Vnus*.[1] There already is a large body of scientific work available on the subject of generic transformation verification mechanisms, and it would be an interesting challenge to investigate ways of incorporating this research into a next-generation *Rotan* framework.

[1] Imagine a compilation system that warns the programmer if a rule just entered is not semantics-preserving. Even if it worked for trivial rules only, this could already be a valuable development aid.

Appendix A

Rule Language Grammar

This appendix specifies the grammar of the *Rule Language*.

Conventions

- *Rule Language* reserved words and literal tokens (including punctuation) are in **`boldface courier`**;
- Non-terminals are in *italics*;
- Grammar meta-symbols are in `courier typeface`;
- The *Rule Language* is case insensitive.

Rule Module Definition

rulemodule:
    $precedences_{opt}$ $rule_{list}$ $properties_{opt}$

precedences:
    `prec engine` *engineId* **`driver`** *driverId* **`begin`** $ruleId_{list}$ **`end.`**
    `prec engine` *engineId* **`begin`** $ruleId_{list}$ **`end.`**
    `prec begin` $ruleId_{list}$ **`end.`**

properties:
    `engine` *engineId* **`reap`**
    `engine` *engineId* **`driver`** *driverId* **`reap`**

rule:
    ruleHeader ruleBody

ruleHeader:
rule *ruleName* **from** *driverName* **in** *engineName* $types_{opt}$ $helptext_{opt}$

types:
types $typeId_{list}$

helptext:
help *stringLiteral*

ruleBody:
begin $applicationMode_{opt}$ *conditionPattern* **->** *actionPattern* **end.**

applicationMode:
once
cont
continuous
reap
reapplication

Condition & Action Patterns

conditionPattern:
conditionNode
expr
list

actionPattern:
actionNode
expr
list

conditionNode:
nonOperatorId $conditionWhere_{opt}$ $code_{opt}$

actionNode:
nonOperatorId $conditionWhere_{opt}$ $code_{opt}$

conditionWhere:
[*conditionClause* **]**

actionWhere:
[*actionClause* **]**

conditionClause:
conditionClause **and** *conditionClause*
conditionClause **or** *conditionClause*

not *conditionClause*
(*conditionClause* **)**
contains *conditionPattern*
attributeId **contains** *conditionPattern*
attributeId **matches** *conditionPattern*
comparand relationOp comparand
rulevar **=** *attribute*
rulevar **=** *rulevar*

comparand:
attribute
$*rulevarId*
stringConst
intConst

actionClause:
actionClause **and** *actionClause*
becomesClause

becomesClause:
attributeId **=** *actionPattern*
$*rulevar* **=** *actionPattern*

code:
string

Expressions

expr:
parenthesisExpr
anyNode
operatorId
operatorId enclosedExprOrTerm
closedExprOrTerm operatorId closedExprOrTerm

parenthesisExpr:
(*expr* **)** *actionWhere* $code_{opt}$
(*expr* **)** *condWhere* $code_{opt}$

closedExprOrTerm:
parenthesisExpr
term
anyNode

exprOrTerm:
expr
term

term:
node
anyNode
leafNode
enclosedList

anyNode:
any $rulevarAssignment_{opt}$ *condWhere* $code_{opt}$
any $rulevarAssignment_{opt}$ *actionWhere* $code_{opt}$

leafNode:
leaf $rulevarAssignment_{opt}$ *actionWhere* $code_{opt}$

rulevarAssignment:
. *rulevar*

Lists

list:
enclosedList
openList

enclosedList:
< $listElement_{list}$ **>** $rulevarAssignment_{opt}$ *actionWhere*$_{opt}$ $code_{opt}$
< $listElement_{list}$ **>** $rulevarAssignment_{opt}$ *condWhere*$_{opt}$ $code_{opt}$

openList:
listElement $listElement_{list}$
$listElement_{list}$
...

listElement:
$listElement_{list}$
...
listElement **|** *listElement*
node
< $listElement_{list}$ **>** $rulevarAssignment_{opt}$ *actionWhere*$_{opt}$ $code_{opt}$
< $listElement_{list}$ **>** $rulevarAssignment_{opt}$ *condWhere*$_{opt}$ $code_{opt}$
< *expr* **>** $rulevarAssignment_{opt}$ *actionWhere*$_{opt}$ $code_{opt}$
< *expr* **>** $rulevarAssignment_{opt}$ *condWhere*$_{opt}$ $code_{opt}$
enclosedList

Tokens

relationOp: one of
== != < > <= >=

keyword: one of
and any begin cont contains continuous driver end engine from help in leaf matches noleaf not once op operator or prec reap reapplication rule types

Lexical Entities

In the productions in this section, whitespace is significant.

stringLiteral:
" *stringChars* **"**

stringChars:
stringChar
stringChars stringChar

stringChar:
any *character* except **"**

identifier:
identifierChars but not a *keyword*

identifierChars:
letter
identifierChars letter
identifierChars digit

letter: one of
_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

intNumber:
$sign_{opt}$ *digits*

digits:
digit
digits digit

digit: one of
1 2 3 4 5 6 7 8 9

sign: one of
+ -

Appendix B

Vnus Grammar

This appendix specifies the grammar of *Vnus*, and is reproduced here from [Ree00b] with the author's permission.

Program

program:
`program` *globalPragmas*$_{opt}$ *declarations block*

Declarations

declarations:
`declarations [` *declaration*$_{list}$ `]`

declaration:
routineDeclaration
variableDeclaration
typeDeclaration

routineDeclaration:
functionDeclaration
procedureDeclaration
externalFunctionDeclaration
externalProcedureDeclaration

variableDeclaration:
globalVariableDeclaration
externalVariableDeclaration
cardinalityVariableDeclaration

localVariableDeclaration
formalVariableDeclaration

typeDeclaration:
recordDeclaration

globalVariableDeclaration:
`globalvariable` *identifier type expression*$_{opt}$ *modifiers*$_{opt}$ *pragmas*$_{opt}$

functionDeclaration:
`function` *identifier formalParameters identifier type modifiers*$_{opt}$ *pragmas*$_{opt}$ *block*

procedureDeclaration:
`procedure` *identifier formalParameters modifiers*$_{opt}$ *pragmas*$_{opt}$ *block*

localVariableDeclaration:
`localvariable` *identifier scopename type expression*$_{opt}$ *modifiers*$_{opt}$ *pragmas*$_{opt}$

formalVariableDeclaration:
`formalvariable` *identifier scopename type modifiers*$_{opt}$ *pragmas*$_{opt}$

cardinalityVariableDeclaration:
`cardinalityvariable` *identifier modifiers*$_{opt}$ *pragmas*$_{opt}$

externalVariableDeclaration:
`externalvariable` *identifier type modifiers*$_{opt}$ *pragmas*$_{opt}$

externalFunctionDeclaration:
`externalfunction` *identifier formalParameters type modifiers*$_{opt}$ *pragmas*$_{opt}$

externalProcedureDeclaration:
`externalprocedure` *identifier formalParameters modifiers*$_{opt}$ *pragmas*$_{opt}$

recordDeclaration:
`record` *identifier* `[` *fieldList* `]` *modifiers*$_{opt}$ *pragmas*$_{opt}$

modifiers:
modifier
modifiers modifier

modifier: one of
`const local unchecked volatile`

block:
`statements` *scopename*$_{opt}$ *pragmas*$_{opt}$ `[` *labeledStatement*$_{list}$ `]`

labeledStatement:
label$_{opt}$ *pragmas*$_{opt}$ *statement*

statement:
imperative
control
parallelization
communication
memoryManagement
support

formalParameters:
`[` *formalParameter*$_{list}$ `]`

formalParameter:
identifier

Flow of Control

control:
while
dowhile
for
if
block
switch
return
valueReturn
goto
throw
rethrow
catch

while:
`while` *expression block*

dowhile:
`dowhile` *expression block*

for:
`for` *cardinalities block*

if:
`if` *expression block block*

switch:
`switch` *expression* `[` $switchCase_{list}$ `]`

return:
`return`

valueReturn:
`return` *expression*

goto:
`goto` *labelName*

throw:
`throw` *expression*

rethrow:
`rethrow`

catch:
`catch` *formalParameter block block*

switchCase:
`(` *intLiteral* `,` *block* `)`
`(` `default` `,` *block* `)`

cardinalities:
`[` $cardinality_{list}$ `]`

cardinality:
identifier `:` *expression*

Parallelization

parallelization:
forall
forkall
fork
foreach
each

forall:
`forall` *cardinalities block*

forkall:
`forkall` *cardinalities block*

fork:
`fork [` $statement_{list}$ `]`

foreach:
`foreach` *cardinalities block*

each:
`each [` $statement_{list}$ `]`

Communication

communication:
barrier
signal
wait
send
receive
blockSend
blockReceive

barrier:
`barrier`

wait:
`wait` *expression*

signal:
`signal` *expression*

send:
`send` *expression expression*

receive:
`receive` *expression location*

blockSend:
`blocksend` *expression expression expression*

blockReceive:
`blockreceive` *expression location location*

Imperative

imperative:
    assignment
    procedureCall
    setSize
    setRoom
    fitRoom

assignment:
    `assign` *location expression*

procedureCall:
    `procedurecall` *routineExpression actualParameters*

setSize:
    `setsize` *location sizes*

setRoom:
    `setroom` *location expression*

fitRoom:
    `fitroom` *location*

routineExpression:
    identifier
    `*` *expression*

Memory Management

memoryManagement:
    delete
    garbageCollect

delete:
    `delete` *expression*

garbageCollect:
    `garbagecollect`

Support Statements

support:
    empty
    print
    println

empty:
`empty`

print:
`print` *actualParameters*

println:
`println` *actualParameters*

Expressions and Locations

location:
identifier
`field` *expression identifier*
selectionLocation
`expressionpragma` *pragmas location*
`*` *expression*
`where` *scopename location*

selectionLocation:
`(` *expression* `,` *selectors* `)`

expression:
literalExpression
accessExpression
constructorExpression
operatorExpression
shapeInfoExpression
miscellaneousExpression

literalExpression:
byteLiteral
shortLiteral
intLiteral
longLiteral
floatLiteral
doubleLiteral
charLiteral
booleanLiteral
stringLiteral
nullLiteral

accessExpression:
identifier
`(` *expression* `,` *selectors* `)`

`field` *expression identifier*
`*` *expression*
`functioncall` *routineExpression actualParameters*

constructorExpression:
`complex` *expression expression*
`shape` *sizes type* `[` $expression_{list}$ `]`
`record` `[` $expression_{list}$ `]`
`&` *location*
`new` *type*
`fillednew` *type expression*

operatorExpression:
`cast` *type expression*
`if` *expression expression expression*
`where` *scopename expression*
`(` *unaryOperator* `,` *expression* `)`
`(` *expression* `,` *binaryOperator* `,` *expression* `)`

shapeInfoExpression:
`ismultidimdist` *expression*
`getblocksize` *expression expression*
`getsize` *expression expression*
`getlength` *expression*
`getroom` *expression*
`sender` *location*
`owner` *location*
`isowner` *location expression*

miscellaneousExpression:
`expressionpragma` *pragmas expression*
`israised` *expression*

selectors:
`[` $selector_{list}$ `]`

selector:
expression

actualParameters:
`[` $actualParameter_{list}$ `]`

actualParameter:
expression

distributions:
`[` $distribution_{list}$ `]`

distribution:
`dontcare`
`block`
`cyclic`
`blockcyclic` *expression*
`replicated`
`collapsed`
`local` *expression*

type:
baseType
`shape` *sizes* $distributions_{opt}$ *type*
`record [` $field_{list}$ `] pointer` *type*
`procedure [` $type_{list}$ `]`
`function [` $type_{list}$ `]` *type*

baseType: one of
`string boolean byte short int long float double char`
`complex`

field:
identifier `:` *type*

sizes:
`[` $size_{list}$ `]`

size:
`dontcare`
expression

Miscellaneous

scopename:
identifier

label:
labelName `:`

labelName:
identifier

globalPragmas:
`pragma` *pragmas*

pragmas:
    `[` *pragma*$_{list}$ `]`

pragma:
    identifier
    identifier `=` *PragmaExpression*

PragmaExpression:
    LiteralPragmaExpression
    NamePragmaExpression
    ListPragmaExpression

LiteralPragmaExpression:
    intLiteral
    floatLiteral
    doubleLiteral
    stringLiteral
    booleanLiteral

NamePragmaExpression:
    identifier
    `@` *identifier*

ListPragmaExpression:
    `(` *PragmaExpressions* `)`

PragmaExpressions:
    empty
    PragmaExpressions PragmaExpression

Tokens

unaryOperator: one of
    `not + ~`

binaryOperator: one of
    `* + - / < << <= <> = > >= >> >>> and mod or xor`

booleanLiteral: one of
    `true false`

nullLiteral:
`null`

keyword: one of
```
and assignment barrier block blockcyclic
blockreceive blocksend boolean byte
cardinalityvariable catch collapsed complex
cyclic declarations delete dontcare double each
empty expression expressionpragma expressions
externalfunction externalprocedure externalvariable
false field fillednew fitroom flag float forall
foreach fork forkall formalvariable function
functioncall garbagecollect getblocksize getlength
getsize globalvariable goto if integer isowner
iteration local localvariable long mod new not null
or owner pointer pragma print println procedure
procedurecall program receive replicated rethrow
return send sender setroom setsize shape short
statements string switch throw true value while xor
```

Lexical Entities

In the productions in this section, whitespace is significant.

byteLiteral:
intNumber byteSuffix

byteSuffix: one of
`b B`

shortLiteral:
intNumber shortSuffix

shortSuffix: one of
`s S`

intLiteral:
intNumber
intNumber intSuffix

intSuffix: one of
`i I`

longLiteral:
    intNumber longSuffix

longSuffix: one of
    `l L`

floatLiteral:
    floatNumber
    floatNumber floatSuffix

floatSuffix: one of
    `f F`

doubleLiteral:
    floatNumber doubleSuffix

doubleSuffix: one of
    `d D`

charLiteral:
    `'` *char* `'`

stringLiteral:
    `"` *stringChars* `"`

stringChars:
    stringChar
    stringChars stringChar

stringChar:
    any *char* except an unescaped '"'

identifier:
    identifierChars but not a *keyword*

identifierChars:
    letter
    identifierChars letter
    identifierChars digit

letter: one of
    `_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

floatNumber:
sign$_{opt}$ *digits floatExponent*
sign$_{opt}$ `.` *digits floatExponent*$_{opt}$
sign$_{opt}$ *digits* `.` *digits*$_{opt}$ *floatExponent*$_{opt}$

floatExponent:
exponentIndicator sign$_{opt}$ *digits*

exponentIndicator: one of
`e E`

intNumber:
sign$_{opt}$ *digits*

digits:
digit
digits digit

digit: one of
`0 1 2 3 4 5 6 7 8 9`

sign: one of
`+ -`

char:
any character
escapedCharacter

escapedCharacter:
`\b`
`\f`
`\n`
`\r`
`\t`
`\"`
`\\`
`\` *digit*
`\` *digit digit*
`\` *digit digit digit*

Appendix C

Vnus Domain Definition

This is the *Tm* domain file for *Vnus*, used to generate an instantiation of the *Rotan* framework specific to the *Vnus* language. All the transformation rules used in the compiler described in Chapter 5 are expressed in terms of identifiers and relationships originating in this file. How this file is used by *Tm* is explained in Chapter 3.

```
|| vnus.ds - Domain definition file for Vnus
||
|| Data structures describing the intermediate
|| programming language Vnus.


NObject == object + (status:String);

Vnusprog    == NObject +
                (pragmas:[Pragma],
                declarations:[Declaration],
                statements:Block);


|| Symbol table entries.

Declaration             == NObject +
                            (name:String,
                            flags:[Modifier],
                            pragmas:[Pragma]);


RoutineDeclaration      == Declaration +
                            (parms:[String]);
BlockRoutineDeclaration == RoutineDeclaration +
                            (body:Block);
DeclFunction            == BlockRoutineDeclaration +
                            (rettype:Type);
DeclProcedure           == BlockRoutineDeclaration + ();
DeclExternalFunction    == RoutineDeclaration +
                            (rettype:Type);
DeclExternalProcedure   == RoutineDeclaration + ();
```

```
VariableDeclaration     == Declaration + ();
DeclGlobalVariable      == VariableDeclaration +
                           (t:Type,
                           init:Expression);
DeclLocalVariable       == VariableDeclaration +
                           (scope:String,
                           t:Type,
                           init:Expression);
DeclFormalVariable      == VariableDeclaration +
                           (scope:String,
                           t:Type);
DeclCardinalityVariable == VariableDeclaration + ();
DeclExternalVariable    == VariableDeclaration +
                           (t:Type);


TypeDeclaration         == Declaration + ();
DeclRecord              == TypeDeclaration +
                           (fields:[Field]);


|| Statements and related structures.

Block               == NObject +
                       (scope:String,
                       statements:[LabeledStatement]);


SwitchCase          == NObject +
                       (body:Block);
SwitchCaseValue     == SwitchCase +
                       (cond:Int);
SwitchCaseDefault   == SwitchCase + ();


Statement           == NObject +
                       (label:String,
                       pragmas:[Pragma]);
LabeledStatement    == Statement + ();


Imperative          == LabeledStatement + ();
SmtAssign           == Imperative +
                       (lhs:Location,
                       rhs:Expression);
SmtProcedureCall    == Imperative +
                       (proc:Expression,
                       parameters:[Expression]);
SmtExpression       == Imperative +
                       (x:Expression);
SmtSetSize          == Imperative +
                       (shape:Location,
                       sizes:[Size]);
SmtSetRoom          == Imperative +
                       (shape:Location,
                       sz:Expression);
SmtFitRoom          == Imperative +
                       (shape:Location);


Control         == LabeledStatement + ();
SmtWhile        == Control +
                   (cond:Expression,
```

```
                    body:Block);
SmtDoWhile      == Control +
                    (cond:Expression,
                    body:Block);
SmtFor          == Control +
                    (cards:[Cardinality],
                    body:Block);
SmtIf           == Control +
                    (cond:Expression,
                    thenbody:Block,
                    elsebody:Block);
SmtBlock        == Control +
                    (body:Block);
SmtSwitch       == Control +
                    (cond:Expression,
                    cases:[SwitchCase]);
SmtReturn       == Control + ();
SmtValueReturn  == Control +
                    (v:Expression);
SmtGoto         == Control +
                    (target:String);
SmtThrow        == Control +
                    (elm:Expression);
SmtRethrow      == Control + ();
SmtCatch        == Control +
                    (elm:String,
                    body:Block,
                    handler:Block);


Parallelization     == LabeledStatement + ();
SmtForall           == Parallelization +
                        (cards:[Cardinality],
                        body:Block);
SmtForkall          == Parallelization +
                        (cards:[Cardinality],
                        body:Block);
SmtFork             == Parallelization +
                        (body:Block);
SmtForeach          == Parallelization +
                        (cards:[Cardinality],
                        body:Block);
SmtEach             == Parallelization +
                        (body:Block);


Semaphores      == LabeledStatement + ();
SmtSignal       == Semaphores +
                    (var:Expression);
SmtWait         == Semaphores +
                    (var:Expression);


Communication   == LabeledStatement + ();
SmtBarrier      == Communication + ();
SmtWaitPending  == Communication + ();
SmtSend         == Communication +
                    (dest:Expression,
                    elm:Expression);
SmtReceive      == Communication +
                    (src:Expression,
                    elm:Location);
SmtBlocksend    == Communication +
                    (dest:Expression,
                    buf:Expression,
```

```
                    nelm:Expression);
SmtBlockreceive == Communication +
                    (src:Expression,
                    buf:Expression,
                    nelm:Expression);
SmtASend        == Communication +
                    (dest:Expression,
                    elm:Expression);
SmtAReceive     == Communication +
                    (src:Expression,
                    elm:Location);
SmtABlocksend   == Communication +
                    (dest:Expression,
                    buf:Expression,
                    nelm:Expression);
SmtABlockreceive == Communication +
                    (src:Expression,
                    buf:Expression,
                    nelm:Expression);


MemoryManagement    == LabeledStatement + ();
SmtDelete           == MemoryManagement +
                        (adr:Expression);
SmtGarbageCollect   == MemoryManagement + ();


Support             == LabeledStatement + ();
SmtEmpty            == Support + ();
SmtPrintSupport     == Support +
                        (argv:[Expression]);
SmtPrint            == SmtPrintSupport + ();
SmtPrintLn          == SmtPrintSupport + ();


Size            == NObject + ();
SizeDontcare    == Size + ();
SizeExpression  == Size +
                    (x:Expression);


Distribution        == NObject + ();
DistDontcare        == Distribution + ();
DistBlock           == Distribution + ();
DistCyclic          == Distribution + ();
DistBC              == Distribution +
                        (blocksize:Expression);
DistReplicated      == Distribution + ();
DistCollapsed       == Distribution + ();
DistLocal           == Distribution +
                        (proc:Expression);

|| Data types.

Type            == NObject + ();

TypeBase        == Type + ();
TypeShape       == Type +
                    (sizes:[Size],
                    distr:[Distribution],
                    elmtype:Type);
TypePointer     == Type +
                    (elmtype:Type);
TypeRecord      == Type +
                    (fields:[Field]);
```

```
TypeNamedRecord == Type +
                    (name:String);
RoutineType     == Type +
                    (formals:[Type]);
TypeProcedure   == RoutineType + ();
TypeFunction    == RoutineType +
                    (rettype:Type);
TypePragmas     == Type +
                    (pragmas:[Pragma],
                     t:Type);

Field           == NObject +
                    (name:String,
                    elmtype:Type);


|| A cardinality - a variable that iterates over a given range, with
|| a given stride. Contrary to many other range notations, the stride
|| should not be negative.
Cardinality     == NObject +
                    (name:String,
                    lowerbound:Expression,
                    upperbound:Expression,
                    stride:Expression,
                    secondaries:[Secondary]);


|| A secondary - a variable that follows a cardinality
Secondary       == NObject +
                    (name:String,
                    lowerbound:Expression,
                    stride:Expression);


Location        == NObject +
                    (pragmas:[Pragma]);
LocName         == Location +
                    (name:String);
LocField        == Location +
                    (rec:Expression,
                    fld:String);
LocFieldNumber  == Location +
                    (rec:Expression,
                    fld:Int);
LocSelection    == Location +
                    (shape:Expression,
                    indices:[Expression]);
LocDeref        == Location +
                    (ref:Expression);
LocWhere        == Location +
                    (scope:String,
                    l:Location);


|| Vnus expressions.

OptExprNone     == LiteralExpression + ();


Expression      == NObject +
                    (pragmas:[Pragma]);


|| Various types of literals.
```

```
LiteralExpression       == Expression +
                               (content:String);
ExprByte        == LiteralExpression + ();
ExprShort       == LiteralExpression + ();
ExprInt         == LiteralExpression + ();
ExprLong        == LiteralExpression + ();
ExprFloat       == LiteralExpression + ();
ExprDouble      == LiteralExpression + ();
ExprChar        == LiteralExpression + ();
ExprBoolean     == LiteralExpression + ();
ExprString      == LiteralExpression + ();
ExprNull        == Expression + ();


AccessExpression        == Expression + ();
ExprName        == AccessExpression +
                    (name:String);
ExprSelection   == AccessExpression +
                    (shape:Expression,
                    indices:[Expression]);
ExprField       == AccessExpression +
                    (rec:Expression,
                    fld:String);
ExprFieldNumber == AccessExpression +
                    (rec:Expression,
                    fld:Int);
ExprDeref       == AccessExpression +
                    (ref:Expression);
ExprFunctionCall    == AccessExpression +
                        (function:Expression,
                        parameters:[Expression]);


ConstructorExpression   == Expression + ();
ExprComplex     == ConstructorExpression +
                    (re:Expression,
                    im:Expression);
ExprShape       == ConstructorExpression +
                    (sizes:[Size],
                    elmtype:Type,
                    arr:[Expression]);
ExprRecord      == ConstructorExpression +
                    (fields:[Expression]);
ExprAddress     == ConstructorExpression +
                    (adr:Location);
ExprNew         == ConstructorExpression +
                    (t:Type);
ExprFilledNew   == ConstructorExpression +
                    (t:Type,
                    init:Expression);


OperatorExpression      == Expression + ();
ExprCast    == OperatorExpression +
                (t:Type,
                x:Expression);
ExprIf      == OperatorExpression +
                (cond:Expression,
                thenval:Expression,
                elseval:Expression);
ExprWhere   == OperatorExpression +
                (scope:String,
                x:Expression);
ExprUnop    == OperatorExpression +
                (optor:Operator,
```

```
                operand:Expression);
ExprBinop   == OperatorExpression +
                (optor:Operator,
                operanda:Expression,
                operandb:Expression);


ExprShapeInfoExpression == Expression + ();
ExprIsMultidimDist  == ExprShapeInfoExpression +
                       (shape:Expression);
ExprGetBlocksize    == ExprShapeInfoExpression +
                       (shape:Expression,
                       dim:Expression);
ExprGetSize         == ExprShapeInfoExpression +
                       (shape:Expression,
                       dim:Expression);
ExprGetLength       == ExprShapeInfoExpression +
                       (shape:Expression);
ExprGetRoom         == ExprShapeInfoExpression +
                       (shape:Expression);
ExprSender          == ExprShapeInfoExpression +
                       (shape:Location);
ExprOwner           == ExprShapeInfoExpression +
                       (shape:Location);
ExprIsOwner         == ExprShapeInfoExpression +
                       (shape:Location,
                       proc:Expression);


AssertExpression        == Expression +
                           (exception:Expression);
ExprNotNullAssert   == AssertExpression +
                       (x:Expression);


MiscellaneousExpression == Expression + ();
ExprPragma              == MiscellaneousExpression +
                           (prs:[Pragma],
                           x:Expression);
ExprIsRaised            == MiscellaneousExpression +
                           (x:Expression);


|| Pragmas

Pragma          == NObject +
                   (name:String);
FlagPragma      == Pragma + ();
ValuePragma     == Pragma +
                   (x:PragmaExpression);


PragmaExpression == NObject + ();

LiteralPragmaExpression == PragmaExpression + ();
NumberPragmaExpression  == LiteralPragmaExpression +
                           (v:String);
StringPragmaExpression  == LiteralPragmaExpression +
                           (s:String);
BooleanPragmaExpression == LiteralPragmaExpression +
                           (b:String);


SymbolPragmaExpression  == PragmaExpression + ();
NamePragmaExpression            == SymbolPragmaExpression +
```

```
                                    (name:String);
ExternalNamePragmaExpression    == SymbolPragmaExpression +
                                    (name:String);


ListPragmaExpression    == PragmaExpression +
                           (l:[PragmaExpression]);


InternalPragma  == Pragma + ();

XPragma         == InternalPragma +
                    (x:Expression);

LocationPragma  == InternalPragma +
                    (loc:Location);

ShapeLocationPragma == InternalPragma +
                        (dist:Distribution,
                        basetype:TypeBase,
                        fulltype:Type,
                        shape:String,
                        distdim:String);

SizePragma      == InternalPragma +
                    (size:Expression);

RenamePragma    == InternalPragma +
                    (from:String,
                        to:Expression);

CardsPragma     == InternalPragma +
                    (cards:[Cardinality]);


Modifier    == NObject + ();
Unchecked   == Modifier + ();
Volatile    == Modifier + ();
Const       == Modifier + ();
Local       == Modifier + ();


TypeString  == TypeBase + ();
TypeBoolean == TypeBase + ();
TypeByte    == TypeBase + ();
TypeShort   == TypeBase + ();
TypeLong    == TypeBase + ();
TypeInt     == TypeBase + ();
TypeFloat   == TypeBase + ();
TypeDouble  == TypeBase + ();
TypeChar    == TypeBase + ();
TypeComplex == TypeBase + ();


Operator        == NObject + ();

OpBin           == Operator + ();
OpAnd           == OpBin + ();
OpOr            == OpBin + ();
OpMod           == OpBin + ();
OpPlus          == OpBin + ();
OpMinus         == OpBin + ();
OpTimes         == OpBin + ();
OpDivide        == OpBin + ();
OpEqual         == OpBin + ();
```

```
OpNotEqual      == OpBin + ();
OpLess          == OpBin + ();
OpLessEqual     == OpBin + ();
OpGreater       == OpBin + ();
OpGreaterEqual  == OpBin + ();
OpXor           == OpBin + ();
OpShiftleft     == OpBin + ();
OpShiftright    == OpBin + ();
OpUShiftright   == OpBin + ();

OpUn            == Operator + ();
OpUNot          == OpUn + ();
OpUPlus         == OpUn + ();
OpUNegate       == OpUn + ();
```

Appendix D

Example Rule

A Rule for Communication Aggregation

This rule is one of the key rules in the *Vnus Rotan* compiler's Optimisation phase, as described in Section 5.4.1. It implements the actual aggregation sub-engine, and is the largest rule in the entire compiler. It is included here to give a flavour for *Rule Language* programming in a more complex form than seen in the examples shown in the previous chapters.

The actual match pattern is still not that complex. The vast bulk of the rule is made up of the specification of the communication template that will replace the matched code. This specification consists of a fairly straightforward block-by-block creation of a new abstract syntax tree (the complexity, of course, went into coming up with the template for this new tree in the first place).

The embedded code used at the end of the rule is also simple in nature: it is mostly concerned with generating unique identifiers for introduced new variables, and with duplicating some of the matched sub-trees.

A full archive of the rules used in the *Rotan Vnus* compiler is available through the Parallel and Distributed Systems group's website at <`http://www.pds.its.tudelft.nl/`>.

```
RULE ca1 FROM ca IN CommunicationAggregation
HELP "Introduce explicit communication around subroutine communication statements"


BEGIN CONT

  Vnusprog.PROG
  [
    CONTAINS DeclProcedure.PD
    [
      $PROCNAME = Name &&
      Parms MATCHES < (...).PARMS String.CB > &&
      CONTAINS Block
      [
        $SCOPE = Scope
        &&
        Statements MATCHES
        <
```

```
        (...).XX
        SmtForeach.F
        [
          Pragmas CONTAINS FlagPragma [ Name == "independent" ] &&
          $C1 = Cards &&
          Cards MATCHES < Cardinality.CARD1 [ $ID = Name && $UPB = Upperbound ]  > &&
          Body
          [
            Statements MATCHES
            <
              SmtAssign.A
              [
                Pragmas CONTAINS FlagPragma.FP [Name == "communication"] &&
                Lhs MATCHES LocName [ $LNAME = Name ] &&
                Rhs MATCHES ExprSelection.RSEL [ $RSELREF = Shape && $RSELSEL = Indices ] &&
                $LHS = Lhs &&
                $RHS = Rhs
              ]
            >
          ]
        ]
        (...).YY
      >.LIST
    ]
  ]
  &&
  Declarations MATCHES < (...).DECLARATIONS > &&
  Declarations CONTAINS Declaration.TEMPODECL
  [
    Name == $LNAME &&
    CONTAINS TypeShape [ Sizes MATCHES < SizeExpression [ $SIZE = X ] > ]  &&
    CONTAINS TypeBase.BUFTYPE &&
    CONTAINS LocationPragma [ $LSEL = Loc ]
  ]
]
->

Vnusprog.PROG
[
  $LIST =
  <
    $XX

    Block [ Scope = $SCOPE_1 && Statements =
    <

      SmtIf
    [
      Cond = ExprBinop
      [
        Optor = OpLess &&
        Operanda = ExprGetSize
          [
          Shape = ExprName [ Name = $BUF ] &&
          Dim = ExprInt [ Content = "0" ]
        ] &&
        Operandb = $SIZE
        ]
      &&
      Thenbody = Block [ Scope = $SCOPE_2 && Statements =
      <
        SmtSetSize [ Shape = LocName [ Name = $BUF] && Sizes = < SizeExpression [ X = $SIZE ] > ]
        >]
    ]

    SmtForeach
    [
      Cards = < Cardinality [ Name = $CV_0 && Upperbound = ExprName [ Name = "numberOfProcessors" ] ] >
        &&
      Body = Block [ Scope = "no scopename" && Statements =
        <
          SmtIf
          [
            Cond = ExprBinop
            [
              Optor = OpNotEqual &&
              Operanda = ExprName [ Name = $CV_0 ] &&
              Operandb = ExprField
            [
              Rec = ExprDeref [ Ref = ExprName [ Name = $CB ] ] &&
              Fld = "_p"
            ]
            ] &&
            Thenbody = Block [ Scope = $SCOPE_4 && Statements =
      <
        SmtForeach
        [
```

```
      Cards = < Cardinality [ Name = $CV_1 && Upperbound = $UPB ] > &&
        Pragmas = < ValuePragma [ Name = "renamecardvar" &&
                                  X = StringPragmaExpression [ S = $ID ] ]
                  > &&
      Body = Block [ Scope = "no scopename" && Statements =
      <
        SmtIf
        [
                Cond = ExprBinop
                [
                  Optor = OpAnd &&
                  Operanda = ExprIsOwner
          [
              Shape = $LHSCLONE &&
              Proc = ExprName [ Name = $CV_0 ]
           ] &&
                Operandb = ExprBinop
           [
             Optor = OpEqual &&
             Operanda = ExprSender
             [
               Shape = LocSelection [ Shape = $RSELREFCLONE && Indices = $RSELSELCLONE ]
             ] &&
               Operandb = ExprField
             [
               Rec = ExprDeref [ Ref = ExprName [ Name = $CB ] ] &&
               Fld = "_p"
             ]
             ]
               ] &&
          Thenbody = Block [ Scope = "no scopename" && Statements =
          <
            SmtAssign
            [
              Lhs = LocSelection
            [
                Shape = ExprName [ Name = $BUF ] &&
              Indices = < ExprName [ Name = $NO1 ] >
            ] &&
            Rhs = $RHSCLONE
            ]
            SmtAssign
          [
            Lhs = LocName [ Name = $NO1 ] &&
              Rhs = ExprBinop
              [
                Optor = OpPlus &&
                Operanda = ExprName [ Name = $NO1 ] &&
                Operandb = ExprInt [ Content = "1" ]
              ]
           ]
         >]
        ]
      >]
    ]

  SmtIf
  [
    Cond = ExprBinop
    [
      Optor = OpNotEqual &&
      Operanda = ExprName [ Name = $NO1 ] &&
        Operandb = ExprInt [ Content = "0" ]
      ] &&
      Thenbody = Block [ Scope = "no scopename" && Statements =
    <
      SmtBlocksend
        [
        Dest = ExprName [ Name = $CV_0 ] &&
        Buf = ExprName [ Name = $BUF ] &&
        Nelm = ExprName [ Name = $NO1 ]
      ]
      >]
          ]
  >]
     ]
    >]
]

  SmtForeach
[
  Cards = < Cardinality [ Name = $CV_2 && Upperbound = ExprName [ Name = "numberOfProcessors" ] ] > &&
    Body = Block [ Scope = "no scopename" && Statements =
  <
      SmtIf
      [
        Cond = ExprBinop
```

```
      [
        Optor = OpNotEqual &&
        Operanda = ExprName [ Name = $CV_2 ] &&
        Operandb = ExprField
      [
        Rec = ExprDeref [ Ref = ExprName [ Name = $CB ] ] &&
        Fld = "_p"
      ]
      ] &&
      Thenbody = Block [ Scope = $SCOPE_5 && Statements =
<

  SmtForeach
  [
    Cards = < Cardinality [ Name = $CV_3 && Upperbound = $UPB ] > &&
    Pragmas = < ValuePragma [ Name = "renamecardvar" &&
                              X = StringPragmaExpression [ S = $ID ] ]
              > &&
    Body = Block [ Scope = "no scopename" && Statements =
      <
      SmtIf
      [
        Cond = ExprBinop
          [
            Optor = OpAnd &&
            Operanda = ExprIsOwner
          [
            Shape = $LHSCLONE2 &&
            Proc = ExprField
            [
              Rec = ExprDeref [ Ref = ExprName [ Name = $CB ] ] &&
              Fld = "_p"
            ]
          ] &&
            Operandb = ExprBinop
          [
            Optor = OpEqual &&
            Operanda = ExprSender
            [
              Shape = LocSelection [ Shape = $RSELREFCLONE2 && Indices = $RSELSELCLONE2 ]
            ] &&
            Operandb = ExprName [ Name = $CV_2 ]
            ]
          ] &&
          Thenbody = Block [ Scope = "no scopename" && Statements =
        <
          SmtAssign
          [
            Lhs = LocName [ Name = $NORCVD ] &&
            Rhs = ExprBinop
            [
              Optor = OpPlus &&
              Operanda = ExprName [ Name = $NORCVD ] &&
              Operandb = ExprInt [ Content = "1" ]
              ]
            ]
        >]
      ]
    >]
        ]

    SmtIf
  [
    Cond = ExprBinop
    [
      Optor = OpNotEqual &&
      Operanda = ExprName [ Name = $NORCVD ] &&
      Operandb = ExprInt [ Content = "0" ]
    ] &&
    Thenbody = Block [ Scope = "no scopename" && Statements =
    <
      SmtBlockreceive
      [
        Src = ExprName [ Name = $CV_2 ] &&
        Buf = ExprName [ Name = $BUF ] &&
        Nelm = ExprName [ Name = $NORCVD ]
      ]
      SmtWaitPending
      SmtForeach
      [
        Cards = < Cardinality [ Name = $CV_4 && Upperbound = $UPB ] > &&
        Pragmas = < ValuePragma [ Name = "renamecardvar" &&
                                  X = StringPragmaExpression [ S = $ID ] ]
                  > &&
        Body = Block [ Scope = "no scopename" && Statements =
        <
          SmtIf
```

```
            [
              Cond = ExprBinop
                [
                  Optor = OpAnd &&
                  Operanda = ExprIsOwner
                  [
                    Shape = $LHSCLONE3 &&
                    Proc = ExprField
                    [
                      Rec = ExprDeref [ Ref = ExprName [ Name = $CB ]] &&
                      Fld = "_p"
                    ]
                  ] &&
                  Operandb = ExprBinop
                [
                  Optor = OpEqual &&
                  Operanda = ExprSender
                  [
                      Shape = LocSelection [ Shape = $RSELREFCLONE3 && Indices = $RSELSELCLONE3 ]
                  ] &&
                  Operandb = ExprName [ Name = $CV_2 ]
                ]
                ]
              &&
              Thenbody = Block [ Scope = "no scopename" && Statements =
              <
                SmtAssign
                [
                  Lhs = $LHSCLONE4
                  &&
                  Rhs = ExprSelection
                  [
                    Shape = ExprName [ Name = $BUF ] &&
                      Indices = < ExprName [ Name = $NO2 ] >
                    ]
                  ]
                SmtAssign
                [
                  Lhs = LocName [ Name = $NO2 ] &&
                  Rhs = ExprBinop
                  [
                    Optor = OpPlus &&
                    Operanda = ExprName [ Name = $NO2 ] &&
                    Operandb = ExprInt [ Content = "1" ]
                  ]
                ]
                >]
            ]
          >]
        ]
      >]
      ]
>] &&

      Elsebody = Block [ Scope = "no scopename" && Statements =
      <
        SmtForeach
        [
          Cards = < Cardinality [ Name = $CV_5 && Upperbound = $UPB ] > &&
          Pragmas = < ValuePragma [ Name = "renamecardvar" &&
                      X = StringPragmaExpression [ S = $ID ] ]
                    > &&
          Body = Block [ Scope = "no scopename" && Statements =
          <
      SmtIf
      [
        Cond = ExprBinop
          [
            Optor = OpAnd &&
            Operanda = ExprIsOwner
            [
              Shape = $LHSCLONE5 &&
              Proc = ExprName [ Name = $CV_2 ]
            ] &&
            Operandb = ExprBinop
          [
            Optor = OpEqual &&
              Operanda = ExprSender
              [
                Shape = LocSelection [ Shape = $RSELREFCLONE4 && Indices = $RSELSELCLONE4 ]
            ] &&
            Operandb = ExprField
              [
                Rec = ExprDeref [ Ref = ExprName [ Name = $CB ]] &&
                Fld = "_p"
              ]
          ]
```

```
                ]
              &&
              Thenbody = Block [ Scope = "no scopename" && Statements =
              <
                SmtAssign
                [
                  Lhs = $LHSCLONE6
                  &&
                    Rhs = ExprSelection
                    [
                    Shape = $RSELREFCLONE5 &&
                      Indices = $RSELSELCLONE5
                    ]
                  ]
                >]
            ]
              >]
            ]
          >]
        ]
      >]
    ]

      SmtWaitPending

    >]
  $YY
  >


  &&
  Declarations =
  <
    $DECLARATIONS
    DeclGlobalVariable
    [
        T = TypeShape
        [
            Elmtype = $BUFTYPE &&
            Sizes = < ExprInt [ Content = "1"] >
        ] &&
        Name = $BUF && Init = ExprNull
    ]

    DeclLocalVariable [ Scope = $SCOPE_4 && T = TypeInt && Name = $NO1 && Init = ExprInt [ Content = "0" ] ]
    DeclLocalVariable [ Scope = $SCOPE_5 && T = TypeInt && Name = $NO2 && Init = ExprInt [ Content = "0" ] ]
    DeclLocalVariable [ Scope = $SCOPE_5 && T = TypeInt && Name = $NORCVD && Init = ExprInt [ Content = "0" ] ]

    DeclCardinalityVariable [ Name = $CV_0 ]
    DeclCardinalityVariable [ Name = $CV_1 ]
    DeclCardinalityVariable [ Name = $CV_2 ]
    DeclCardinalityVariable [ Name = $CV_3 ]
    DeclCardinalityVariable [ Name = $CV_4 ]
    DeclCardinalityVariable [ Name = $CV_5 ]
  >
]
{
      // Embedded C++ code
  $SCOPE_1 = new String(idGenerator->Unique("__scope_"));
  $SCOPE_2 = new String(idGenerator->Unique("__scope_"));
  $SCOPE_4 = new String(idGenerator->Unique("__scope_"));
  $SCOPE_5 = new String(idGenerator->Unique("__scope_"));

  $CV_0 = new String(idGenerator->Unique("__ca3_cv_"));
  $CV_1 = new String(idGenerator->Unique("__unique_cv_"));
  $CV_2 = new String(idGenerator->Unique("__ca3_cv_"));
  $CV_3 = new String(idGenerator->Unique("__unique_cv_"));
  $CV_4 = new String(idGenerator->Unique("__unique_cv_"));
  $CV_5 = new String(idGenerator->Unique("__unique_cv_"));

  $NO1 = new String(idGenerator->Unique("__ca3_no1_"));
  $NO2 = new String(idGenerator->Unique("__ca3_no2_"));

  $NORCVD = new String(idGenerator->Unique("__ca3_noRcvd_"));
  $BUF = new String(idGenerator->Unique("__ca3_buf_"));

  $LHSCLONE = (Location *)$LSEL->Clone();
  $LHSCLONE2 = (Location *)$LSEL->Clone();
  $LHSCLONE3 = (Location *)$LSEL->Clone();
  $LHSCLONE4 = (Location *)$LSEL->Clone();
  $LHSCLONE5 = (Location *)$LSEL->Clone();
  $LHSCLONE6 = (Location *)$LSEL->Clone();

  $RHSCLONE = (Expression *)$RHS->Clone();

  $RSELREFCLONE = (Expression *)$RSELREF->Clone();
  $RSELSELCLONE = (List *)$RSELSEL->Clone();
```

```
    $RSELREFCLONE2 = (Expression *)$RSELREF->Clone();
    $RSELSELCLONE2 = (List *)$RSELSEL->Clone();
    $RSELREFCLONE3 = (Expression *)$RSELREF->Clone();
    $RSELSELCLONE3 = (List *)$RSELSEL->Clone();
    $RSELREFCLONE4 = (Expression *)$RSELREF->Clone();
    $RSELSELCLONE4 = (List *)$RSELSEL->Clone();
    $RSELREFCLONE5 = (Expression *)$RSELREF->Clone();
    $RSELSELCLONE5 = (List *)$RSELSEL->Clone();
  }

END.
```

Bibliography

[Aik98] A. Aiken, M. Fändrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98), Kyoto, Japan*. Mar 1998.

[Amd67] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, vol. 30, pp. 483–485. ACM, AFIPS Press, Reston, Va., 1967.

[ASC02] ASCI. The distributed ASCI supercomputer 2 (DAS-2) website, 2002. `http://www.cs.vu.nl/das2/`.

[Bax97] I. Baxter and C. Pidgeon. Software changes through design maintenance. In *Proceedings of the International Conference on Software Maintenance*. IEEE, IEEE press, 1997.

[Ber98] J. A. Bergstra and P. Klint. The discrete time ToolBus — a software coordination architecture. *Science of Computer Programming*, vol. 31(2–3):pp. 205–229, 1998.

[Bra01] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, vol. 2027:pp. 365–370, 2001.

[Bra02] M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Computational Complexity*, pp. 143–158. 2002.

[Bre91] L. C. Breebaart, E. M. Paalvast, and H. J. Sips. The Booster approach to annotating parallel algorithms. In *1991 International Conference on Parallel Processing*, vol. II, pp. 276–277. 1991.

[Bre92] L. C. Breebaart, E. M. Paalvast, and H. J. Sips. A rule based transformation sytem for parallel languages. In *Third Workshop on Compilers for Parallel Computers*, p. 1321. ACPC/TR 928, Vienna, Austria, 1992.

[Bre95] L. C. Breebaart, P. F. G. Dechering, A. B. Poelman, J. A. Trescher, J. P. M. de Vreught, and H. J. Sips. The Booster language: Syntax and static semantics. Computational Physics report series CP-95-02, Delft University of Technology, 1995.

[Bri98] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th Int. World Wide Web Conf.* 14–18 Apr 1998.

[Cla99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, 1999.

[Cor02] J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 10.2*. Legasys Corp., Ontario, Canada, Apr 2002.

[Cra02] Cray Inc. Corporate website, 2002. `http://www.cray.com/`.

[Cza00] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Dec96] P. F. G. Dechering, L. C. Breebaart, F. Kuijlman, C. v. Reeuwijk, and H. J. Sips. A sound and simple semantics of the `forall` statement within the *V-nus* compiler framework. In *6th Workshop on Compilers for Parallel Computers (CPC'96)*, pp. 59–71. Aachen, Germany, Dec 1996.

[Dec97a] P. F. G. Dechering, L. C. Breebaart, F. Kuijlman, and C. v. Reeuwijk. Semantics and implementations of a generalized `forall` statement for parallel languages. In *International Parallel Processing Symposium*, pp. 542–548. IEEE, Geneva, Apr 1997.

[Dec97b] P. F. G. Dechering, L. C. Breebaart, F. Kuijlman, C. v. Reeuwijk, and H. J. Sips. A generalized forall concept for parallel languages. In *Proc. 9th Intl. Workshop, Languages and Compilers for Parallel Computing, LNCS 1239*, pp. 605–607. 1997.

[Dec98] P. F. G. Dechering. *Semantics for Compiling Data Parallelism*. Ph.D. thesis, Delft University of Technology, Delft, Mar 1998.

[Dia94] S. L. Diamond. Architecture Neutral Distribution Format (ANDF). *IEEE Micro*, vol. 14(6):pp. 73–76, Dec 1994.

[Don90] C. Donnelly and R. Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Bison Version 1.12 edn., Dec 1990.

[Elm96] S. Elmohamed. Examples in high performance fortran. Website, 1996. `http://www.npac.syr.edu/projects/cpsedu/summer98summary/examples/hpf/hpf.html`.

[Emm89] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG — A generator for efficient back ends. In B. Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pp. 227–237. ACM Press, Portland, OR, USA, Jun 1989.

[Gei94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Network Parallel Computing*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1994.

[GMD02] GMD (German National Research Center For Information Technology). Catalog of compiler construction tools, 1996–2002. `http://catalog.compilertools.net/`.

[Hal96] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, vol. 29(12):pp. 84–89, 1996.

[Has96] S. B. Hassen and H. E. Bal. Integrating task and data parallelism using shared objects. In *Proceedings 10th ACM International Conference on Supercomputing*, pp. 317–324. ACM Press, Philadelphia, PA, USA, May 1996.

[Hat91] P. J. Hatcher and M. J. Quinn. *Data Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[Hed01] G. Hedin and E. Magnusson. JastAdd — a Java-based system for implementing frontends. In *Proceedings First Workshop on Language Descriptions, Tools and Applications*. ENTCS, Elsevier, 2001.

[Hee02] J. Heering and P. Klint. Rewriting-based languages and systems. In T. Group, editor, *Term Rewriting Systems*, chap. 15. Cambridge University Press, Mar 2002.

[HPF97] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edn., Feb 1997.

[Hud96] S. E. Hudson. *CUP Parser Generator for Java*. Graphics Visualization and Usability Center, Georgia Institute of Technology, Nov 1996.

[Joh75] S. C. Johnson. Yacc — yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[Jon91] E. de Jong. Vista definition report. Tech. Rep. 91 ITI 261, TNO Institute of Applied Computer Science (ITITNO), Delft, The Netherlands, February 1991.

[Jon01] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In *Proceedings First Workshop on Language Descriptions, Tools and Applications*. ENTCS, Elsevier, 2001.

[Kui01] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Proceedings First Workshop on Language Descriptions, Tools and Applications*. ENTCS, Elsevier, 2001.

[Les75] M. E. Lesk. LEX — A lexical analyzer generator. *Computing Science TR*, vol. 39, Oct 1975.

[Leu97] A. Leung. *Prop Language Reference Manual*. Courant Institute of Mathematical Sciences, `http://valis.cs.nyu.edu:8888/~leunga/`, 2.3.4 edn., Apr 1997.

[Lin99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, Apr 1999.

[Mar94] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, vol. 37(5):pp. 58–70, 1994.

[Mar98] F. Martin. PAG — an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, vol. 2(1):pp. 46–67, 1998.

[Mic02] Microsoft. .NET website, 2002. `http://www.microsoft.com/net/`.

[Mor01] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern-matching compiler. In *Proceedings First Workshop on Language Descriptions, Tools and Applications*. ENTCS, Elsevier, 2001.

[Muc93] V. B. Muchnick, A. V. Shafarenko, and C. D. Sutton. *F*-code and its implementation: a portable software platform for data parallelism. *The Computer Journal*, vol. 36(8):pp. 712–722, 1993.

[Nel00] G. Nelan. *App — An Algebraic Typing and Pattern Matching Preprocessor for C++*, 2000. User Manual.

[Paa91] E. M. Paalvast, H. J. Sips, and L. C. Breebaart. Booster: a high-level language for portable parallel algorithms. *Applied Numerical Mathematics: Transactions of IMACS*, vol. 8:pp. 177–192, Sep 1991.

[Paa92] E. M. Paalvast. *Programming for Parallelism and Compiling for Efficiency*. Ph.d., Delft University of Technology, Jun 1992.

[Par95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software —Practice and Experience*, vol. 25(7):pp. 789–810, Jul 1995.

[Pax95] V. Paxson. *Flex, version 2.5: A fast scanner generator*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 2.5 edn., Mar 1995.

[Pem82] S. Pemberton and M. C. Daniels. *Pascal Implementation — The P4 Compiler*. Ellis Horwood, Chicester, UK, 1982.

[Per96] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Lecture Notes in Computer Science 1132. Springer Verlag, 1996.

[Por01] The Portland Group. *PGHPF Reference Manual*, 2001.

[Ree92] C. v. Reeuwijk. Tm: a code generator for recursive data structures. *Software — Practice and Experience*, vol. 22(10):pp. 899–908, Oct 1992.

[Ree96] C. v. Reeuwijk, W. Denissen, H. J. Sips, and E. M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 7(9):pp. 897–914, Sep 1996.

[Ree00a] C. v. Reeuwijk. Template manager reference manual. PDS Technical Report PDS-2000-003, Delft University of Technology, May 2000. `http://www.pds.twi.tudelft.nl/reports/2000/PDS-2000-003/`.

[Ree00b] C. v. Reeuwijk. The Vnus language specification, version 2.1. PDS Technical Report PDS-2000-002, Delft University of Technology, May 2000. `http://www.pds.twi.tudelft.nl/reports/2000/PDS-2000-002/`.

[Ree01] C. v. Reeuwijk. Spar 1.5 language specification. PDS Technical Report PDS-2001-003, Delft University of Technology, Oct 2001. http://www.pds.twi.tudelft.nl/reports/2000/PDS-2001-003/.

[Ree03a] C. v. Reeuwijk. The Timber compiler. Website, 1998–2003. http://www.pds.twi.tudelft.nl/timber.

[Ree03b] C. v. Reeuwijk. Rapid and robust compiler construction using template-based metacompilation. In *Proc. 12th International Conference on Compiler Construction (CC 2003), to be published*. Springer Verlag, Warsaw, Poland, Apr 2003.

[Rod96] B. Rodriguez, L. Hart, and T. Henderson. Parallelizing operational weather forecast models for portable and fast execution. *Journal of Parallel and Distributed Computing*, vol. 37(2):pp. 159–170, 1996.

[SET02] SETI@Home. The search for extraterrestrial intelligence, 2002. http://setiathome.ssl.berkeley.edu/.

[Sni96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA., 1996.

[Ste92] M. R. van Steen and H. J. Sips. The ParTool project: Development of a parallel programming environment. In R. Perrott, editor, *Software for Parallel Computers*. Chapman & Hall, London, 1992.

[Tji86] S. Tjiang. Twig reference manual. Comp. Sci. Tech. Rep. 120, AT&T Bell Laboratories, Jan 1986.

[Tum02] E. Tumenbayar. Linux SMP HOWTO, 2002. http://ouray.cudenver.edu/~etumenba/smp-howto/.

[Vis01a] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, Utrecht, The Netherlands, 2001.

[Vis01b] E. Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, vol. 57/2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, Utrecht, The Netherlands, May 2001.

[Vis02] E. Visser. The program tranformation Wiki, 2002. http://www.program-transformation.org/.

[Web02] Webgain. Javacc website, 2002. http://www.webgain.com/products/java_cc/.

[Wil02] P. Wilders, A. Ecer, N. Satofuka, J. Periaux, and P. Fox, editors. *Parallel Computational Fluid Dynamics 2001*. North-Holland, April 2002.

[Wol95] L. Wolters, G. Cats, and N. Gustafsson. Data-parallel numerical weather forecasting, 1995.

Summary

Rule-based Compilation of Data-parallel Programs

Certain computational problems are too big or complex for a conventional single-processor system to solve in a reasonable amount of time. In such cases, parallel computing is an approach that might be considered instead: by having several processors work on the problem simultaneously, the total execution time can be brought down to acceptable levels.

Unfortunately, writing explicitly parallel programs is a skill that does not come very naturally to human beings. It is difficult to correctly keep track of the different parallel program threads, and the need for communication and synchronisation between these programs only adds to the complexity.

This is why much research has gone into the creation of high-level parallel programming languages in which the user is shielded from (too much) explicit parallelism. These languages allow the programmer to pretend to a considerable extent that they are working in a conventional, single-thread model of computation.

This has been a quite successful approach as far as the applications programmer is concerned, but under the hood the complexity and the programming difficulties have not gone away, but have merely been shifted around. The high-level programs must still somehow be converted to explicitly parallel applications, only now it is the compilation software, not the programmer, that is responsible for achieving this, preferably in a manner that will lead to highly efficient target code. Consequently, compilation techniques for parallel programming languages have also become a fruitful area for research. Both the compilation algorithms themselves, and the way in which they can be specified by the compiler builder are of interest.

This thesis investigates particular technologies that can make the task of writing compilers for parallel programming languages more manageable and less error-

prone. A compiler-generator framework called *Rotan* is described. *Rotan* can be used to create a programmable compiler for a programming language. This compiler can be programmed by the compiler builder in a high-level, pattern-matching transformation language called the *Rule Language*. This turns the compiler from the conventional static 'black box' systems into a more dynamic, open transformation system, that allows easy and modular experimentation, debugging, and extension of compilation and optimisation algorithms.

Chapter 1 (*Introduction*) contains a general introduction to the thesis and discusses the research question.

In Chapter 2 (*Data Parallelism*) we give a brief general introduction to parallel programming, followed by a closer explanation of the data-parallel programming model that will be the focus for the remainder of the thesis. In data-parallel programming, the programmer specifies the distribution of data over the processors. It is left to the compiler to choose and implement the efficient distribution of the actual computations over the processors, and this is precisely where the challenge lies.

In Chapter 3 (*Compiler Construction Tools*) existing compilation models for sequential and parallel programming are described, and an overview of existing compilation tools and approaches is given. This chapter also contains a review of general-purpose transformation systems. The programmable compiler framework called the *Rotan* system is proposed as a means of obtaining the levels of flexibility, expressive power, and maintainability a compiler for (data-)parallel programming languages requires.

Chapter 4 (*The Rule Language*) introduces the *Rule Language*, the rule-based transformation language that forms one of the key components in the *Rotan* framework. The *Rule Language* allows the compiler builder to implement translations and optimisations by specifying them as high-level transformations on the parse tree of a source program. This chapter explains the syntax and gives an informal operational semantics of the *Rule Language* as implemented in the current *Rotan* prototype.

Chapter 5 (*A Rotan Compiler for Vnus*) describes the major test case for the *Rotan* system: an implementation of a semi-automatically parallelising compiler for the *Vnus* language. *Vnus* is a programming language used as an intermediate format in the compilation process of higher-level (data-)parallel programming languages; the word 'semi-automatic' signifies the fact that the compiler has help during translation, in the form of the data distributions specified by the user. The parallelisation and communication schemes used in this compiler are discussed, and examples of their implementation as rewrite rules are given.

Chapter 6 (*Experimental Results*) presents a number of case studies in which the *Vnus* compiler described in Chapter 5 is applied to data-parallel *Vnus* programs. Since a higher abstraction level is generally associated with a decrease in efficiency, this chapter investigates the extent to which this holds true for the target code generated by the *Rotan Vnus* compiler. The performance results of these programs are then compared to those achieved by other compilers. As it

turns out, there is indeed a performance penalty, but one that remains within the bounds of acceptability.

Chapter 7 (*Evaluation*) evaluates the experiences with the general *Rotan* system both in terms of its own design criteria as well as in comparison to a different compilation system in use at the Delft University of Technology. The evaluations show that the expected advantages of a high-level, rule-based compilation system are real and significant.

Chapter 8 (*Conclusion*) finishes with a summary and discussion of the presented research topics, concluding with some suggestions for future research directions.

Leo Breebaart

Samenvatting

Regelgebaseerde Compilatie van Data-parallelle Programma's

Er zijn problemen die te groot of te complex zijn om door een conventioneel uniprocessor systeem opgelost te worden binnen een redelijke tijdsduur. In zulke gevallen is parallel rekenen een aanpak die overwogen zou kunnen worden: door verschillende processoren tegelijkertijd aan het probleem te laten werken kan de totale uitvoeringstijd teruggebracht worden tot acceptabele niveaus.

Helaas is het zo, dat het schrijven van expliciet parallelle programma's een bekwaamheid is die de mens niet van nature gegeven is. Het is moeilijk om zicht te houden op de verschillende parallelle programmalijnen, en de noodzaak tot communicatie en synchronisatie tussen deze lijnen laat de complexiteit alleen nog maar toenemen.

Dit verklaart waarom er zo veel onderzoek is verricht naar het creëren van hogere parallelle programmeertalen die de gebruiker afschermen van (te veel) expliciet parallellisme. Deze talen staan het de programmeurs toe in hoge mate te doen alsof zij werken met een conventioneel, niet-parallel programmeermodel.

Deze aanpak is redelijk succesvol geweest voor zover het de applicatieprogrammeur betreft, maar onderhuids zijn de complexiteit en de programmeerproblemen niet zozeer verdwenen, als wel verschoven. De in de hogere programmeertaal uitgedrukte programma's moeten nog steeds, op de één of andere manier, omgezet worden naar expliciet parallelle applicaties. Het is nu echter de compilatie-software, en niet de programmeur, die de verantwoordelijkheid heeft dit te bereiken, en wel liefst op een zodanige wijze dat er hoogst efficiënte doelcode geproduceerd wordt. Compilatietechnieken voor parallelle programmeertalen zijn derhalve ook een dankbaar onderzoeksgebied geworden. Zowel de compilatie-algoritmes zelf, als de manier waarop deze gespecificeerd kunnen worden door de compilerbouwer zijn hier van belang.

Dit proefschrift onderzoekt specifieke technologieën waarmee de taak van het

schrijven van compilers voor parallelle programmeertalen beter beheersbaar en minder foutgevoelig wordt. Het beschrijft een compiler-generator raamwerk genaamd *Rotan*. *Rotan* kan gebruikt worden om een programmeerbare compiler voor een programmeertaal te maken. Deze compiler kan door de compilerbouwer geprogrammeerd worden in een hogere, patroonherkennende transformatietaal, de *Rule Language* geheten. Hiermee verandert de compiler van een conventioneel, statisch 'black box' systeem in een meer dynamisch, open transformatiesysteem, waarin gemakkelijk en modulair experimenteren, debuggen, en uitbreiden van compilatie- en optimalisatie-algoritmen mogelijk zijn.

Hoofdstuk 1 (*Inleiding*) bevat een algemene inleiding tot het proefschrift, en bespreekt de onderzoeksvraag.

In Hoofdstuk 2 (*Data-parallellisme*) wordt begonnen met het geven van een korte algemene inleiding tot het parallel programmeren, gevolgd door een nadere uitleg over het data-parallelle programmeermodel waar in dit proefschrift verder de nadruk op zal liggen. Data-parallel programmeren houdt in dat de programmeur de distributie van data over de processoren specificeert. Het is aan de compiler om dan voor de daadwerkelijke berekeningen een efficiënte distributie over de processoren te kiezen en te implementeren, en dit is nu precies waar de uitdaging ligt.

In Hoofdstuk 3 (*Gereedschappen voor Compilerconstructie*) worden bestaande compilatiemodellen voor sequentieel en parallel programmeren beschreven, en wordt een overzicht gegeven van bestaande compilatiegereedschappen. Dit hoofdstuk bevat ook een bespreking van generieke transformatiesystemen. Voor het compileren van (data-)parallelle programmeertalen zijn bepaalde niveaus van flexibiliteit, uitdrukkingskracht, en onderhoudbaarheid nodig, en het programmeerbare compilatieraamwerk *Rotan* wordt geïntroduceerd als gereedschap om deze niveaus mee te bereiken.

Hoofdstuk 4 (*De Rule Language*) beschrijft de *Rule Language*, de op herschrijfregels gebaseerde transformatietaal die een van de sleutelcomponenten in het *Rotan*-raamwerk is. De *Rule Language* stelt de compilerbouwer in staat om vertalingen en optimalisaties te implementeren door deze op hoog niveau te specificeren als transformaties op de programmaboom van de broncode. Dit hoofdstuk beschrijft de syntax en geeft een informele operationele semantiek van de *Rule Language* zoals deze is geïmplementeerd in het huidige *Rotan*-prototype.

Hoofdstuk 5 (*Een Rotan-Compiler voor Vnus*) gaat dieper in op een grote testcase voor het *Rotan*-systeem: een implementatie van een semi-automatisch parallelliserende compiler voor de taal *Vnus*. *Vnus* is een programmeertaal die gebruikt wordt als een tussenformaat in het compilatieproces van (data-)parallelle programmeertalen van hoger niveau; het begrip 'semi-automatisch' slaat op het feit dat de compiler hulp krijgt bij het vertalen in de vorm van de door de gebruiker gespecificeerde data-distributies. De parallellisatie en communicatie-schema's die de compiler gebruikt worden besproken, en voorbeelden van hun implementatie als herschrijfregels worden gegeven.

Hoofdstuk 6 (*Experimentele Resultaten*) presenteert een aantal voorbeelden

waarin de in Hoofdstuk 5 beschreven *Vnus* compiler wordt toegepast op data-parallelle *Vnus* programma's. Aangezien een hoger abstractieniveau in het algemeen leidt tot een verlies aan efficiëntie, onderzoekt dit hoofdstuk in hoeverre dit ook geldt voor de doelcode die gegenereerd wordt door de *Rotan Vnus* compiler. De prestaties van deze programma's worden dan vergeleken met die welke bereikt worden door andere compilers. Het blijkt dat er inderdaad enig prestatieverlies optreedt, maar in een mate die binnen de grenzen van het acceptabele blijft.

Hoofdstuk 7 (*Evaluatie*) evalueert de ervaringen met het generieke *Rotan* systeem zowel in termen van de oorspronkelijke ontwerpcriteria als in vergelijking met een ander compilatiesysteem dat in gebruik is aan de Technische Universiteit Delft. De evaluaties tonen aan dat de verwachte voordelen van een op hoog niveau gecodeerd, op herschrijfregels gebaseerd compilatiesysteem reëel en significant zijn.

Hoofdstuk 8 (*Conclusie*) eindigt met een samenvatting van en discussie over het gepresenteerde onderzoek, resulterende in enige suggesties voor toekomstig onderzoek.

Leo Breebaart

Curriculum Vitae

Leo Breebaart was born in Amsterdam on September 27, 1966. He moved with his family to Suriname, South America, when he was seven years old.

After returning to the Netherlands to finish secondary school in 1984, he studied Applied Computer Science at Leiden University. He graduated in 1989 and went on to become a PhD student (in Dutch: *Assistent in Opleiding* or AIO) with the Faculty of Applied Physics at the Delft University of Technology as a member of the *ParTool* project.

In the years that followed, he worked as a systems administrator for the Computational Physics group at Delft University, fulfilled his military service in the Dutch Army, worked as a researcher in the Parallel and Distributed Systems group at Delft University, and was researcher in the Simulators group of the TNO Physics and Electronics Laboratory (TNO-FEL) in The Hague, before starting an extended sabbatical, during which he finally found time to write up his PhD research.

He is currently employed as a software engineer for Science & Technology BV in Delft.

Colophon

This manuscript was typeset by the author with the LaTeX 2_ε Documentation System on a PC running Debian GNU/Linux (unstable).

Text editing was done in *GNU Emacs* using the AUCTeX package. The illustrations and graphs were created with *Dia* and *gnuplot*, respectively. The cover layout was produced with *Adobe Illustrator*.

The body type is 10 point Computer Modern Roman. Chapter and section titles are in various sizes of Adobe Helvetica-Narrow Bold. The monospace typeface used for program code is Adobe Courier.

The final output was converted to a PostScript file, which was then transferred to film and printed on 90 grams Crossbow paper by Printing Establishment Ponsen & Looijen bv, Amsterdam.