Compilation of Term Rewriting Systems

Compilation of Term Rewriting Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr P.W.M. de Meijer
ten overstaan van een door het college van dekanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op maandag 16 september 1996 te 10:30 uur

door
Jasper Fredrikus Theodorus Kamperman
geboren te Eindhoven

Promotor: prof.dr P. Klint.
Co-promotor: dr H.R. Walters.

Acknowledgements

First and foremost, I wish to thank my promotor Paul Klint, without whose audacity to hire a physicist for Computer Science research, I would never have started this book. It has always been a pleasure to work in the stable and friendly atmosphere Paul has created in his research group, and I particularly value the amount of freedom he has allowed me in finding my way through Computer Science. I would also like to thank Anneke Groos, for bringing me in contact with Paul.

It is impossible to imagine what this thesis would have looked like without my co-promotor Pum Walters. I thank him for all his wonderful ideas, his patience during innumerable discussions, and all the energy he invested in making the Abstract Rewrite Machine, the equational language EPIC and the accompanying tools into what they are now. To me, the cooperation with Pum is a model of how rewarding teamwork can be.

Many thanks also go to Jan Heering, especially for his advice on Chapters 3 and 6. His meticulous reading and the ensueing comments have been vital for the structure and quality of those chapters. For his comments on Chapter 6, I also want to thank Piet Rodenburg.

Geographically far away, but conceptually close, John Field has been an invaluable source of insightful comments, on both Chapter 6 and Chapter 7. Wan Fokkink, Bas Luttik, and Jaco van de Pol were a great help for Chapter 3.

I want to express a particularly warm gratitude to the members of the reading committee, Zena Ariola, Jan Bergstra, Pieter Hartel, Rinus Plasmeijer and Henk Sips, for investing their time in understanding my thesis and for providing positive criticism. In a different role, I want to thank Pieter Hartel for pulling (yes, he used a *strong* rope) me and Pum into his 'pseudoknot' benchmarking project.

I will probably fail to be complete in an attempt to mention the most important other colleagues that have made life agreeable during the last five years, but I will try anyway: Krzysztof Apt, Huub Bakker, Jaco de Bakker, Annette Bleeker, Doeko Bosscher, Mark van den Brand, Mieke Bruné, Arie van Deursen, Casper Dik, Dinesh, Steven Eker, Job Ganzevoort, Robert Giegerich, David Griffioen, Annius Groenink, Jan Friso Groote, Lynda Hardman, Jaap Henk Hoepman, Bart Jacobs, Steven Klusener, Wilco Koorn, Jan Willem Klop, Henri Korver, Robert van Liere, Emma van der Meulen, Leon Moonen, Dimitri Naidich, Pieter Olivier, Femke van Raamsdonk, Jan Rekers, Arno Siebes, Frank Teusink, Chris Thieme, François Thomasset, Frank Tip, Susan Üsküdarlı, Leonie van der Voort, and Eelco Visser.

Finally, I want to thank my family and friends for always believing in me, and enduring my lack of interest during the past few months. This holds most for Wieneke, whose love and understanding helped me to finish this thesis in a happy mood.

Partial support for this work was received from the European Community under the ESPRIT project 5399 (Compiler Generation for Parallel Machines – COMPARE).

Contents

Chapter 1

Introduction

1.1 Problem Statement

Term (graph) rewriting systems are becoming increasingly important for the implementation of theorem provers [GL91, Fra94, GG91, KZ89, Bou94], verification tools, algebraic specifications [EM85, BHK89a, Vis96, HM93], compiler generators [ESL89], language prototyping [vDHK96], program analyzers [BDHF96] and functional programming languages [PvE93]. This implies a need for techniques enabling fast term and graph rewriting.

Ideally, these techniques should be available in the form of generally applicable software components, which would make the implementation of the tools mentioned above a process of combining components, rather than reimplementing techniques.

To date, this ideal situation seems far away. Even though *techniques* are often shared, *sharing of code* is rare. We see the following causes of this state of affairs:

- There are often incompatibilities between tools, e.g., tools with different type systems are hard to combine.
- Usually, tools do not implement a single feature, but they provide some practical combination of features. As a result, a collection of tools implementing a set of features required to solve a particular problem usually contains much overlap in functionality, leading to inefficient use of resources.
- Many tools are specialized towards particular uses or hardware platforms, and accordingly are less generally applicable.
- Usually, the only well-documented interface concerns the textual, human-readable, level. Therefore, combining tools usually results in both loss of information (e.g., the result of type analysis is lost), and introduction of redundancy (e.g., concrete syntax representations are produced from abstract syntax representations and then parsed into abstract syntax representations).

In this thesis, we attempt to improve upon this situation. Over the years, we have arrived at the following, admittedly bold, thesis, which has developed into the central theme of our work:

> *All software tools for symbolic processing that do not crucially depend on side-effects can* efficiently *be executed as single-sorted left-linear term rewriting systems (TRSs), restricted to innermost rewriting and specificity ordering.*

A few notes should be made about this statement:

- Without the requirement of *efficiency*, this would be trivial, because a term rewriting system (or even a string rewriting system) can be used to emulate a Turing machine.
- The restrictions of single sortedness, left-linearity, innermost rewriting and specificity ordering are imposed to facilitate the implementation of a core system. Alternative type-systems, nonlinear rewriting and alternative strategies are to be implemented on top of this core system.
- Following a general convention, we will mean rewriting of *directed acyclic graphs* (DAGs) when we write 'implementation of term rewriting' in this thesis. Strictly speaking, this is not correct (see [KKSdV93]), but for a sufficiently large class of applications, DAG rewriting *is* equivalent.

Obviously, the implementation of TRSs is central to the exploration of our theme. In a natural way, this leads to the topics discussed in this thesis:

- Eventually, TRSs must be executed on some concrete machine. In order to avoid dependence on the instruction set of a concrete machine, we designed *minimal term rewriting systems* (MTRSs), a subclass of TRSs. By a trivial interpretation, an MTRS can be viewed as a program for an *abstract* machine, the Abstract Reduction Machine (ARM). An ARM program can either be interpreted by a program running on a concrete machine (an ARM-interpreter), or it can be translated into a program running directly on a concrete machine. ARM and MTRSs are defined in Chapter 3.
- In order to make general TRSs executable as ARM programs, they must be translated into the subclass of MTRSs. This translation is elaborated in Chapter 3, and a concrete implementation is given in Chapter 5.
- In order to use TRSs as a (target) programming language, their operational semantics and (abstract) syntax must be specified rigorously. As a step towards such a definition, in Chapter 4, we define the programming language EPIC, which can be used to express Term Rewriting Systems. The operational semantics of EPIC is innermost rewriting with syntactic specificity ordering. The emphasis is on abstract syntax, but one concrete syntax is given as well.

- Even though innermost rewriting is most efficient in time per reduction step, it is not always a desirable strategy. In some cases, innermost rewriting leads to an excessive total number of rewrites (or even non-termination), which could have been avoided by selectively choosing non-innermost redexes. In many functional programming languages, *lazy evaluation* is used to avoid innermost rewriting. In Chapter 6, we define *lazy rewriting* as lazy evaluation in the context of Term Rewriting Systems, and show how this can be implemented given an implementation of innermost rewriting.

- Information-preserving interfacing between tools is complicated by the fact that the external (textual) representation of terms and graphs is usually voluminous, or even infinite (because of cycles), and that producing and analyzing such texts is costly. In Chapter 7, we present a technique to exchange general graphs between tools at a low cost, approaching one byte per node for large graphs (under mild assumptions).

1.2 Brief History

The research reported in this thesis supports the implementation of the ASF+SDF meta-environment, an environment for the interactive specification of programming environments. The ASF+SDF meta-environment is based on ASF+SDF [BHK89a], a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF.

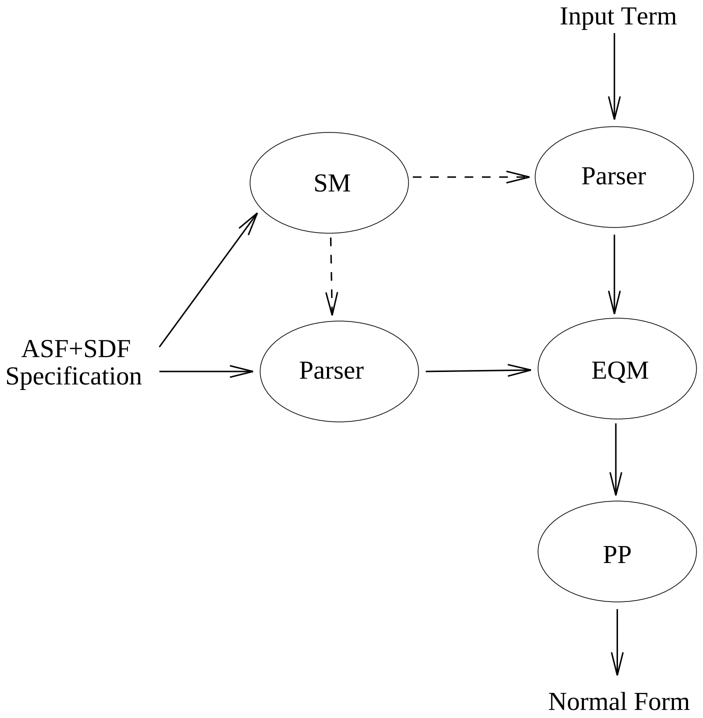


Figure 1.1: Operation of the LeLisp implementation of the meta-environment

When our research started, the ASF+SDF meta-environment operated as sketched in Figure 1.1. An ASF+SDF specification was provided by a user, parsed by a parser that was itself generated from the specification by the Syntax Manager (SM), and registered with the Equation Manager (EQM). Then, an input term (parsed by the same parser) was presented to EQM, which reduced it to normal form, and sent it to the Pretty Printer (PP) to produce a readable version of the normal form.

It should be noted that the division into components of Figure 1.1 is conceptual, in reality the components were glued together as a large collection of LeLisp code. Because LeLisp was not freely available, it became an obstacle to the widespread use of programming environments specified in ASF+SDF.

With a speed of about 2000 rewrites per second, the Equation Manager was significantly faster than the rewrite engines of other systems, such as OBJ and Larch (with topspeeds of at most a few hundred rewrites per second), but not fast enough for many realistic applications, which require 100.000 rewrites per second or more. After some initial experiments, we were convinced that the speed could be dramatically improved by introducing an abstract rewriting machine (ARM). When implemented in a widely available language, ARM could also be used to remove the dependency on LeLisp. These considerations lead to the development of ASF2C, a compiler that could be used to produce stand-alone C programs from ASF+SDF specifications.

In Figure 1.2, we show the operation of ASF2C as a replacement for EQM. Because SDF allows general context-free grammars, and because ASF+SDF specifications cannot be described by one fixed grammar, we could not use existing parser technology other than the Syntax Manager incorporated in the ASF+SDF system. Therefore, an extra tool, SM2RN1 (written in LeLisp by Jan Rekers), was used to produce external representations of specifications and terms from the abstract syntax trees delivered by the parser. Because the initial prefix format (called RN1, for Rule Notation 1) turned out to be lengthy, GEL was used for compression.

From an RN1 specification, ASF2C produced a C program, in which ARM instructions were expressed as C macros. The C program was compiled into an executable, which took a term (in prefix or GEL representation) on standard input and produced a normal form of that term on standard output (in prefix or GEL representation). Because the input terms could be described by a fixed grammar, parsers for input terms could be generated using widely available parser generators (by restricting to the classes of grammars supported by those generators). With the help of ASF2C, Mark van den Brand and Eelco Visser constructed a pretty-printer generator [vdBV96]. With these ingredients, it became possible to generate tools that would function without the support of the meta-environment.

Using the ASF+SDF meta-environment and ASF2C, a language for use in COMPARE[1], fSDL (*full Structure Definition Language*), was designed and a prototype compiler for fSDL was implemented. To date, the language fSDL (but not the prototype compiler) is being used for the development of commercially available compilers by erstwhile COMPARE

---

[1] Acronym for the ESPRIT project COMpiler generation for PARalEl machines.

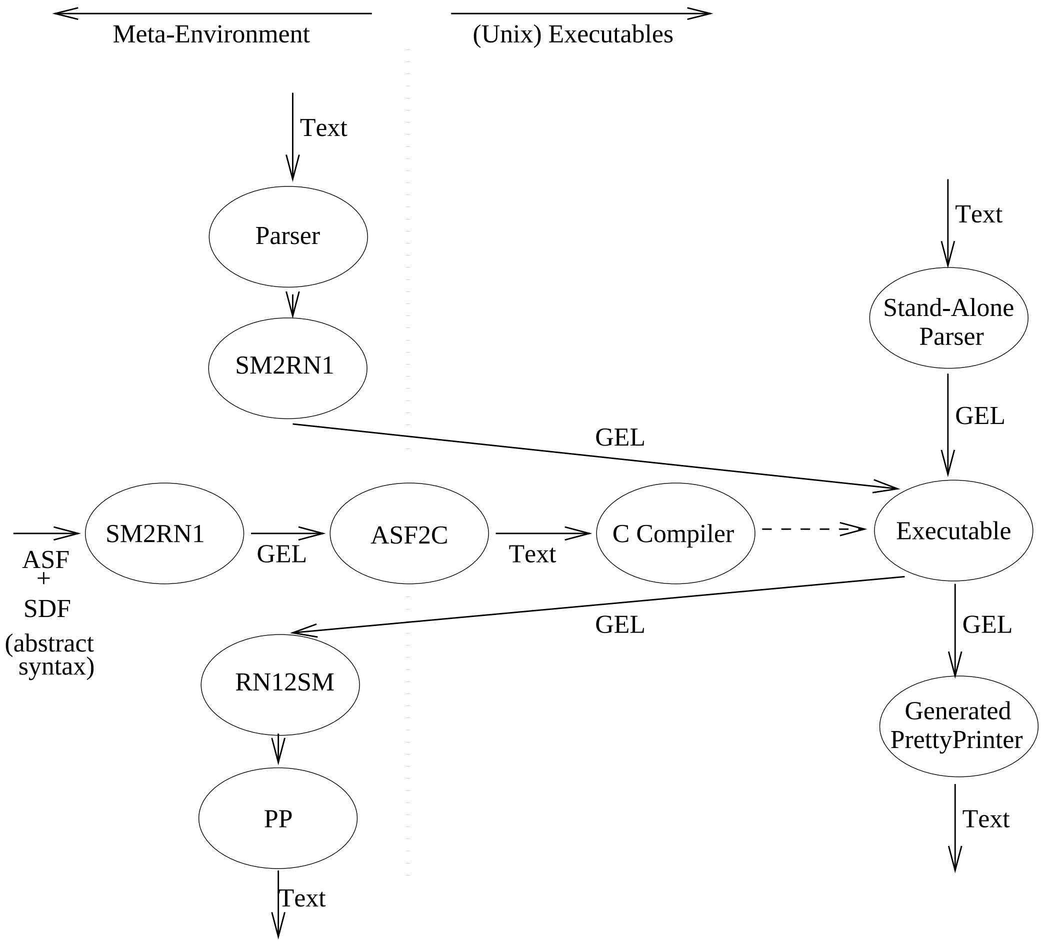


Figure 1.2: ASF2C as replacement for EQM

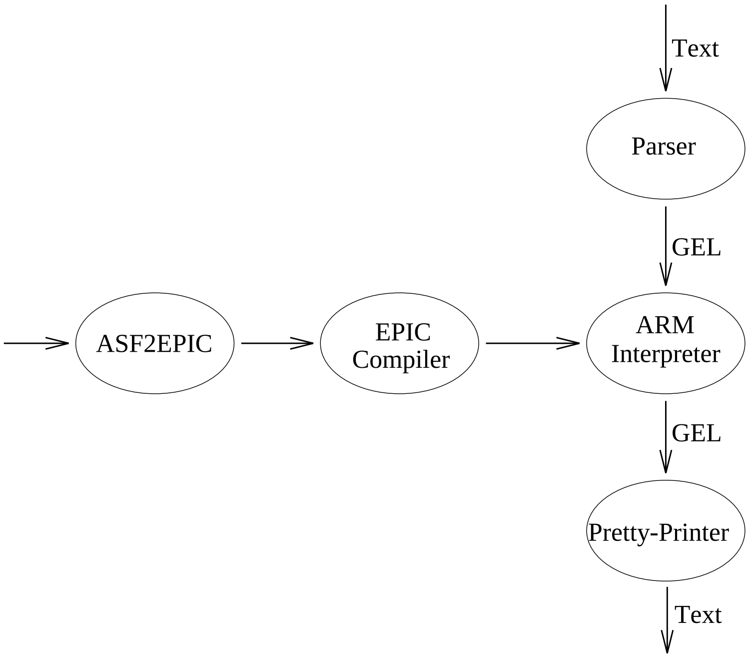


Figure 1.3: The use of EPIC to replace ASF2C

partners.

However, this and other projects uncovered several shortcomings of ASF2C:

- The C compilation producing the final executables took enormous amounts of time.
- Having started as an experimental project, ARM and ASF2C did not feature *garbage collection*, causing huge memory requirements.
- Term rewriting is essentially a process without side effects. Consequently, executables generated by ASF2C could not consume input or produce output incrementally.

On the positive side, these projects showed that the specification and implementation of tools using ASF+SDF was feasible. Furthermore, disregarding memory requirements and compilation time, the tools produced in this way had a useful performance/functionality rate. Therefore, we deemed it worthwile to continue this research, and to address the problems above.

In the mean time, we had realized that many features that were hard-wired in the version of ARM used by ASF2C, could in fact be simulated by transformations of TRSs. This greatly simplified the design and implementation of the ARM interpreter, and also limited the amount of error-prone C programming and maintenance. This lead us to the design of EPIC and the version of ARM described in this thesis. Their use is sketched in Figure 1.3.

The picture is similar to the picture in Figure 1.2, with the exceptions that part of the translation has been shifted to the component ASF2EPIC, and that the C compiler is not needed anymore, because ARM programs are interpreted directly (the C compiler is used exactly once to produce the ARM interpreter from its sources).

The speed attained by the EPIC/ARM combination, in the order of 500.000 rewrites per second, is sufficient for realistic applications. E.g., the EPIC compiler compiles itself in a few minutes.

Finally, we would like to note that, even though our main concern was to carry out an implementation project, the need to rigorously define and understand syntax and semantics of the interfaces between components has resulted in various theoretical excursions, which form the larger part of this thesis.

1.3 Summary of Contributions

Mainly because of its extreme simplicity, the Abstract Rewriting Machine (ARM) is a significant contribution to the existing techniques for implementation of TRSs.

By virtue of their interpretation as ARM programs, and by virtue of the fact that they are, in a practical sense, minimal, Minimal Term Rewriting Systems (MTRSs) are an important subclass of TRSs.

The description in Chapter 3 of the transformation of TRSs into MTRS is simple, correct, and leads to efficient code. We have not found a comparable combination of concerns in earlier publications discussing compilation of TRSs.

The language EPIC and its supporting tools provide a sound basis for the re-engineering of the ASF+SDF system, and other systems based on Term (DAG) Rewriting.

Our definition of *lazy rewriting* in Chapter 6 extends the theory of term rewriting with a concept analogous to that of *lazy evaluation* in functional languages. Our definition is less operational in character than that of lazy evaluation in functional programming languages. Furthermore, an implementation technique is presented that requires only a minimal extension of technology for innermost rewriting, such as ARM. Both the definition of lazy rewriting and the transformation for lazy rewriting on eager machinery further the understanding of the interaction between lazy evaluation and pattern matching.

Finally, the Graph Exchange Language GEL and its supporting tools can efficiently be used to build graph-processing software from graph-processing components.

1.4 Bibliographic Notes

There are not many ideas in this thesis which can really be claimed by a single person, a consequence of the fact that the work described was done in a group, and more specifically, most of it in close collaboration with Pum Walters. However, with the exception of Chapter 4 (see below), the literal expression of this thesis is mine.

Chapter 3 is an extended version (incorporating material of [KW96] and [FvdP96]) of a paper that was presented at the 11th Workshop on Abstract Data Types in Oslo, and will be published as part of the selected papers, [KW95b].

Although Pum Walters is the primary author of Chapter 4, it is included in this thesis since it is essential for the understanding of Chapter 5.

Chapter 6 discusses lazy rewriting on eager machinery, and is an improved version of [KW95a] (Piet Rodenburg rewrote the definition of lazy rewriting, and provided crucial proofs).

Chapter 7 discusses the graph exchange language GEL. This chapter was published as CWI report [Kam94a], and was presented at the Dagstuhl workshop on 'Functional Programming in the real world' [Kam94b, GH94]. Software and documentation are available on WWW, via URL `http://www.cwi.nl/epic`.

Finally, we would like to mention here some related publications that we did not incorporate in this thesis. In [KW93a], a predecessor of ARM was published, fSDL was published as [KDW94], and in [WK96a], we published a technique for incorporating I/O in Term Rewriting Systems. A system description of EPIC, (defined in Chapter 4), Abstract Rewriting Machine, and supporting tools is published as [WK96b]. The system is available on WWW, via URL `http://www.cwi.nl/epic`.

Chapter 2

Term Rewriting

In this chapter, we mainly follow [Mid90] and [Klo92], except for the notation of paths, which is taken from [DJ90]. In Section 2.1 we review *abstract reduction systems*, in Section 2.2 we specialize to *term rewriting systems*, in Section ?? we give a definition of residuals and in Section 2.4 we discuss some restrictions on TRSs.

2.1 Abstract Reduction Systems

Definition 1 *An* abstract reduction system (ARS) *is a pair* (A, R) *consisting of a set* A *and a binary relation* R *on* $A \times A$*.* R *is called a* rewrite relation *or* reduction relation*.*

We will write R^+ for the transitive closure of R, and R^* for the transitive-reflexive closure of R. For the next definitions, we will assume an ARS (A, R).

Definition 2 $a \in A$ *is* a normal form *or* a *is* in normal form *(with respect to R) if there does not exist a* $b \in A$ *such that* aRb*. We call* b *a* normal form of $a \in A$ *if* aR^*b *and* b *is a normal form.*

Definition 3 $a \in A$ *is* terminating *with respect to* R *if there does not exist an infinite reduction* $aRa_1Ra_2R\cdots$*. We call the ARS* (A, R) terminating *if all* a *in* A *are terminating with respect to* R*.*

Definition 4 $a \in A$ *is* weakly terminating *with respect to* R *if there is a normal form* b *such that* aR^*b*. We call the ARS* (A, R) weakly terminating *if all* a *in* A *are weakly terminating with respect to* R*.*

Instead of (weakly) terminating, the terminology *(weakly) normalising* is often used. Note that termination implies weak termination, but not vice versa.

Definition 5 $a \in A$ *is* locally confluent *with respect to* R *if for each pair of reductions* aRa_1 *and* aRa_2*, there exists an* a_3 *such that* $a_1R^*a_3$ *and* $a_2R^*a_3$*. We call* (A, R) *locally confluent if* R *is locally confluent for each* a *in* A*.*

Definition 6 *$a \in A$ is* confluent *with respect to R if for each pair of reductions aR^*a_1 and aR^*a_2, there exists an a_3 such that $a_1R^*a_3$ and $a_2R^*a_3$. We call (A, R) confluent if R is confluent for each a in A.*

Definition 7 *$a \in A$ has the property of* unique normal forms *if a has at most one normal form. We say that (A, R) has the* unique normal forms *property if each a in A has the* unique normal forms *property.*

Proposition 1 *Confluence $\Rightarrow$ Unique normal forms.*
Proof See [Klo92] ■

2.2 Term Rewriting Systems

Abstract reduction systems are specialized to *Term Rewriting Systems* (TRSs) by taking the elements of A to be *terms over a signature* Σ, and R to be the *rewrite relation* induced by a set of *rewrite rules*. Below, we will subsequently define the notions *signature*, *rewrite rule* and *rewrite relation*.

Definition 8 *A* signature Σ *consists of:*

1. *a countably infinite set $\mathcal{V}$ of* variables*: $x, y, \ldots$;*
2. *a non-empty set $\mathcal{F}$ of* function symbols*: $f, g, \ldots$, each with a fixed* arity *(≥ 0), which is the number of arguments the function requires. We denote the arity of a function symbol f by $|f|$. Function Symbols with arity 0 are called* constants.

Definition 9 *The set $T(\Sigma)$ of* terms over Σ *is the smallest set satisfying*

1. *$\mathcal{V} \subset T(\Sigma)$,*
2. *for all $f \in \mathcal{F}$ with arity n, and $t_1, \ldots, t_n \in T(\Sigma)$, we have $t = f(t_1, \ldots, t_n) \in T(\Sigma)$. The t_i ($1 \leq i \leq n$) are called the* arguments *of t.*

We will write $var(t)$ for the set of variables occuring in t. A term is called *linear* if no variable occurs more than once in it. Occasionally, we will abbreviate a sequence $t_1, \ldots, t_n$ to $\vec{t}$, and write $\mid \vec{t} \mid$ for n. We generalize this to empty sequences, for which we have $\mid \vec{t} \mid = 0$. We write $ofs(f(\vec{t})) = f$ for the *outermost function symbol* f of a term $f(\vec{t})$.

Definition 10 *A* path in a term *is a sequence of positive integers, separated by periods. By $t|_p$, we denote the* subterm *of t at path p.*

For example, if $t = f(g, h(f(y, z)))$, then $t|_{2.1}$ is the first subterm of t's second subterm, which is $f(y, z)$. We write $p \in s$ if p is a valid path in s (i.e., indicates a subterm of s), and $p_1 \leq p_2$ if p_1 is a prefix of p_2 (i.e., $\exists p_3 : p_2 = p_1.p_3$). We write $p|q$ iff neither $p \leq q$ nor $q \leq p$. The empty path (referring to root) is written as ε.

We write $t[s]_p$ for the term resulting from the replacement at p of $t|_p$ in t by s. Following [HL91b], we write $\mathcal{O}(s)$ for the *occurences* of s, that is $\{p \mid p \in s\}$.

Definition 11 *A set of paths P is called* prefix-reduced *if there are no pairs $p, p' \in P$ such that p is a prefix of p'. A set S of subterms of s is called* prefix-reduced with respect to s *if there is a prefix-reduced set of paths $\{p_1, \ldots, p_n\}$ such that $S = \{s|_{p_1}, \ldots, s|_{p_n}\}$.*

Definition 12 *A* context $C[\ldots]$ *is a term containing at least one occurrence of a special constant $\Box$. If $C[\ldots]$ is a context with n occurrences of $\Box$ and $t_1, \ldots, t_n$ are terms then $C[t_1, \ldots, t_n]$ is the result of replacing from left to right the occurrences of $\Box$ by t_1 through t_n. A context containing exactly one occurrence of $\Box$ is denoted by $C[]$.*

We say that t is a subterm of $C[t]$, written as $t \subseteq C[t]$.

Definition 13 *A* substitution *is a (total) map $\sigma : T(\Sigma) \mapsto T(\Sigma)$ satisfying*

$$\forall f \in \mathcal{F} \ : \ \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).$$

By convention, we often write t^σ for $\sigma(t)$. A substitution that differs from the identity only on variables will be called *trivial*.

Definition 14 *A* rewrite rule *over $T(\Sigma)$ is a pair of terms written as $s \rightarrow t$ with $s, t \in T(\Sigma)$, which satisfies the following conditions:*

1. *The left hand side s should not be a single variable.*
2. *All variables occurring in the right hand side should also occur in the left hand side, i.e., $var(t) \subseteq var(s)$.*

We write $lhs(l \rightarrow r) = l$ for the *left hand side* l of a rule $l \rightarrow r$, and $rhs(l \rightarrow r) = r$ for the *right hand side* r of a rule $l \rightarrow r$.

Definition 15 *A* term rewriting system $\mathcal{R}$ *is a pair (Σ, R) consisting of a signature Σ and a set of rewrite rules R over $T(\Sigma)$.*

Definition 16 *A* rewrite step *in (Σ, R) is a pair of terms $s \rightarrow t$, such that there is a substitution $\sigma \in T(\Sigma) \times T(\Sigma)$, a position $p \in s$ and a rule $u \rightarrow v$, such that:*

1. *$s|_p = u^\sigma$, and*
2. *$t = s[v^\sigma]_p$.*

The union of all rewrite steps is called the *rewrite relation* $\rightarrow_{\mathcal{R}}$. Since the subscript $\mathcal{R}$ is usually clear from the context, it is omitted. The overloading of $\rightarrow$ is by convention. The subterm u^σ is referred to as *redex* (for reducible expression); the subterm v^σ, as *reduct*.

A position and a rewrite rule completely determine a rewrite step. If we want to be specific about the redex path p and the rule $l \rightarrow r$, we write $s \xrightarrow{p}_{(l \rightarrow r)} t$.

The rewrite relation is closed under contexts, i.e., if $s \rightarrow t$, then for all $C[]$, $C[s] \rightarrow C[t]$.

We call a sequence $s = s_1 \rightarrow s_2 \rightarrow \ldots$ a *rewrite sequence*.

Definition 17 *A function symbol f is called a* defined *function symbol if there is a rule $f(t_1, \ldots, t_n) \rightarrow r$. We will call a function* fully defined *it does not occur in normal forms. A function symbol c is called a* constructor *symbol if there is a normal form in which it occurs, and a* free constructor *if it is not a defined symbol.*

A TRS $\Re$ is called *left-linear* if the lhs of every rewrite rule in $\Re$ is linear.

Proposition 2 *Confluence and termination of TRSs are undecidable properties [DJ90].*

2.3 Residuals

Definition 18 (Overlap) *Let $r_1 : l \rightarrow r$ and $r_2 : g \rightarrow d$ be rewrite rules. If there exists a context $C[]$, a non-variable term s, and a substitution σ such that $l = C[s]$ and $s^\sigma = g^\sigma$, then g is said to* overlap with l.

We say there is *overlap between a rule r and a TRS T* iff either the lhs of r overlaps with the lhs of a rule of T, or there is a rule of T whose lhs overlaps with the lhs of r.

A TRS is called *orthogonal* if it is left-linear, and there is no overlap between lhs's of the rules.

Following [HL91b], we write $\mathcal{R}(s)$ for the set of paths to redexes in s.

Given a rewrite step $A : s \overset{p_A}{\rightarrow}_{(l \rightarrow r)} t$ and $p \in \mathcal{R}(s)$, where there is no overlap between l and the lhs of the rule of p, we define the set $p \backslash A$ of *residuals* or *descendants* of p by A as a subset of $\mathcal{O}(s)$:

$$p \backslash A = \begin{cases} \emptyset & \text{if } p = p_A; \\ \{p\} & \text{if } p | p_A \text{ or } p \leq p_A; \\ \{p_i p_n p_r \mid r|_{p_n} = x\} & \text{if } p = p_i p_m p_r \text{ and } l|_{p_m} = x \in \mathcal{V}. \end{cases}$$

For rewrite sequences, we define $p \backslash A$ by

$$\begin{cases} p \backslash \epsilon = & \{p\} \\ p \backslash AB = & \bigcup_{p_a \in p \backslash A} p_a \backslash B \end{cases}$$

For orthogonal systems (where there is no overlap at all) these definitions specialize to the ones given in [HL91b].

A *development* is a rewrite sequence in which only residuals of redex occurrences that are present in the initial term are contracted.

A redex is said to be *needed* if a residual of it is contracted in every sequence leading to a normal form.

2.4 Restrictions on TRSs

In general, a term may contain many redexes. A rewriting *strategy* determines which of these is chosen. A well-known strategy is *rightmost innermost*, which chooses the rightmost redex that does not contain another redex.

In *priority* rewrite systems (PRSs) [BBKW89], the rules are (partially) ordered, and a rule may be applied only if there are no applicable rules (i.e., even after reduction of subterms) with higher priority. PRSs are very expressive, but their operational semantics can be problematic, see [vdP96]. For our purposes, a weaker notion suffices, which we will call *syntactic priority.* In a TRS with syntactic priority, the decision whether a rule is applicable is made without considering reductions of subterms.

The ordering we will use is *syntactic specificity ordering.*

Definition 19 (syntactic specificity ordering) *A rule* $r_1 : s_1 \to t_1$ *is called* more specific *than a rule* $r_2 : s_2 \to t_2$*, written as* $r_1 > r_2$*, when there exists a non-trivial substitition* σ *such that* $s_2^\sigma = s_1$*.*

In [BBKW89], *specificity ordering* implies that all ambiguities are between terms that are ordered according to specificity, which we do not demand for *syntactic specificity ordering.* We will call two terms s, t (or rules with lhs's s, t) *mutually exclusive* if there is no term u with $u > s \wedge u > t$.

Definition 20 (Most general term and rule) *A term of the form* $f(\vec{x})$ *(where* $\vec{x}$ *is a sequence of distinct variables) is called* most general term*, and a rule with a most general lhs is called a* most general rule*.*

A TRS is called *sufficiently complete* if defined functions do not appear in normal forms (see [GH78]). Sufficient completeness is undecidable. We will use a simpler notion:

Definition 21 *A TRS is* simply complete *if every defined function has a most general rule.*

It is clear that simple completeness implies sufficient completeness.

Term *graph* rewriting (see [BvEJ$^+$87]), where the objects that are rewritten are graphs, can be seen as a restriction of rewriting with infinite terms, as discussed in [KKSdV93]. An implementation of term rewriting can be turned into an implementation of graph rewriting by taking care that the sharing expressed by graphs is retained.

Chapter 3

Minimal Term Rewriting Systems

A simple, efficient and correct compilation technique for left-linear Term Rewriting Systems (TRSs) is presented. TRSs are compiled into *Minimal Term Rewriting Systems* (MTRSs), a subclass of TRSs. MTRS rules have such a simple form that they can be seen as instructions for an extremely simple Abstract Rewriting Machine (ARM).

The correctness of the compilation from a TRS into an MTRS is demonstrated by showing that the MTRS *simulates* the TRS. The MTRS simulates neither too much (*soundness*) nor too little (*completeness*), nor does it introduce unwarranted infinite sequences (*termination conservation*). The compiler and its correctness proof are largely independent of the reduction strategy.

3.1 Introduction

A standard technique for speeding up the execution of a program in a formal (programming) language is *compilation* of the program into the language of a concrete machine (e.g., a microprocessor). In compiler construction (c.f. [ASU86]), it is customary to use an *abstract machine* as an abstraction of the concrete machine. On the one hand, this allows hiding details of the concrete machine in a small part of the compiler, and thus an easy reimplementation on other concrete machines. On the other hand, a good design of the abstract machine enables a simple mapping from source language into abstract machine language.

A compiler consists of zero or more transformations in the semantic domain of its source language, followed by a mapping to a lower-level language. This is repeated until the level of the concrete machine is reached. Because they take place in one domain, the source-to-source transformations are easier to grasp semantically than the mappings to lower levels. In this chapter, we present a compilation technique for TRSs which stays entirely within the well-known source language domain.

TRSs are compiled into *Minimal Term Rewriting Systems* (MTRSs), a syntactic restriction of TRSs. Rules occurring in MTRSs have very simple patterns, and there is little

$$\begin{array}{rcl} plus(zero, x) & \rightarrow & x \\ plus(succ(x), y) & \rightarrow & succ(plus(x, y)) \end{array}$$

Figure 3.1: A well-known TRS defining addition on natural numbers

difference between variable configurations on both sides. As a result, the application of an MTRS rule is an elementary operation. By a modest change of perspective, an MTRS can be seen as a program for the Abstract Rewriting Machine (ARM), which is in turn easily implemented on a concrete machine.

The idea to express pattern matching of TRSs in the language of TRSs itself was inspired by [Pet92], where pattern matching of ML is expressed in ML itself. That paper does not contain a correctness proof, and the algorithm is formulated in a less formal way than our algorithm. The resulting pattern match code appears to have the same complexity as the code produced by our algorithm.

The idea to include a correctness proof is taken from [HG94], in which steps towards a provably correct compiler for OBJ3 are taken. Their compiler is less geared towards efficiency than ours. Furthermore, our compiler and its correctness proof are largely independent of reduction strategy.

In the remainder of this chapter, we proceed as follows. In Section 3.2, we give an informal overview of the compilation process. Subsequently, we present MTRSs and their interpretion as machine language for ARM in Section 3.3. In Section 3.6, we define what it means for one TRS to simulate another, and in Section 3.7 we show how a simulating MTRS can be constructed for any TRS. Finally, we discuss related work in Section 3.8, and we draw conclusions in Section 3.9.

3.2 Informal overview

We will first describe the compilation of TRSs into MTRSs informally, starting at the end. First we assume innermost rewriting with specificity ordering, and show the plausibility of interpreting MTRSs with this strategy as machine code for a simple abstract machine. Then we indicate how complicated pattern matching can be expressed in MTRS terms.

3.2.1 A Simple Example of an MTRS

Consider Figures 3.1 and 3.2. In Figure 3.1 a well-known TRS defining addition on natural numbers is shown, and in Figure 3.2, an MTRS is depicted (the relation to the TRS in Figure 3.1 is discussed below).

We note the following properties of the MTRS in Figure 3.2 (a formal definition of MTRSs is given in Section 3.3):

- In every rule, at most three function symbols occur, of which at most two on either

$$
\begin{array}{rcl}
zero & \rightarrow & r(zero_c) \\
succ(x') & \rightarrow & r(succ_c(x')) \\
plus(x', x'') & \rightarrow & plus^S(x', x'') \\
plus(zero_c, z') & \rightarrow & plus\_zero(z') \\
plus(succ_c(y'), z') & \rightarrow & plus\_succ(y', z') \\
plus\_zero(y') & \rightarrow & y' \\
plus\_succ(y', y'') & \rightarrow & succ(plus(y', y'')) \\
plus^S(x', x'') & \rightarrow & r(plus_c(x', x'')) \\
r(x) & \rightarrow & x
\end{array}
$$

Figure 3.2: Addition as an MTRS

side. Even the SKI calculus ([Klo92]), which is minimal in the number of rules (3), and in the total number of function symbols (4: S, K, I, and ap), needs 7 function symbols in its most complicated rule $(\mathrm{ap}(\mathrm{ap}(\mathrm{ap}(\mathrm{S}, x), y), z) \rightarrow \mathrm{ap}(\mathrm{ap}(x, z), \mathrm{ap}(y, z)))$.

- In every rule, the variables of the lhs occur also in the rhs, in exactly the same order (this is already an accidental property of the original system). In MTRSs, a (number of consecutive) variable(s) may be missing on the rhs, or a variable may be duplicated.

- Every function is either fully defined or a free constructor.

- The outermost function symbol of a right hand side is always a defined symbol.

- In all rules with the same function symbol outermost on the lhs which have a second function symbol in the lhs, this symbol occurs in the same argument position of the outermost symbol.

Important for the interpretation as a machine language, is the fact that the 'action' (adding, changing or deleting function symbols or variables) performed by application of a rule is 'local', i.e. restricted to a number of consecutive arguments and the outermost function symbol.

The MTRS in Figure 3.2 may be used to *simulate* (in a sense made precise in Section 3.6) the computations that can be made using the TRS in Figure 3.1. Consider for example the term $plus(succ(zero), succ(zero))$. In the system of Figure 3.1 this term can be rewritten to $succ(succ(zero))$. Innermost rewriting in the MTRS of Figure 3.2 (preferring more specific rules over general rules) yields the term $succ_c(succ_c(zero_c))$. The latter normal form can be said to *simulate* the former by assuming a *simulation map* $\mathcal{S}$ (see Section 3.6.4) defined as $\mathcal{S}(zero_c) = zero$ and $\mathcal{S}(succ_c(x)) = succ(\mathcal{S}(x))$.

Note that $plus_c$ is used to simulate normal forms in which the function symbol $plus$ occurs (taking $\mathcal{S}(plus_c(x, y)) = plus(\mathcal{S}(x), \mathcal{S}(y))$).

3.2.2 Interpreting MTRSs as Abstract Machine Code

By a slight change of perspective, the MTRS in Figure 3.2 can be interpreted as a program for the Abstract Reduction Machine (ARM) (the resemblance to assembly code is intended):

$$\begin{array}{ll}
\mathit{zero}: & \mathbf{build}(\mathit{zero}_c, 0); \mathbf{goto}(r)\\
\mathit{succ}: & \mathbf{build}(\mathit{succ}_c, 1); \mathbf{goto}(r)\\
\mathit{plus}: & \mathbf{match}(\mathit{zero}_c, \mathit{plus\_zero});\\
 & \mathbf{match}(\mathit{succ}_c, \mathit{plus\_succ});\\
 & \mathbf{goto}(\mathit{plus}^S);\\
\mathit{plus\_zero}: & \mathbf{recycle};\\
\mathit{plus\_succ}: & \mathbf{cpush}(\mathit{succ}); \mathbf{goto}(\mathit{plus});\\
\mathit{plus}^S: & \mathbf{build}(\mathit{plus}_c, 2); \mathbf{goto}(r);\\
r: & \mathbf{recycle},
\end{array}$$

where the intuitive meaning of the instructions is as follows:

build. Builds a term from a function symbol and a specified number of arguments taken from the *argument* stack

match. Determines whether the term on top of the argument stack is built from a specified funtion symbol. If this is the case, the term on top of the argument stack is replaced by its arguments, and execution continues at the specified label. If the term on top of the argument stack is built from another symbol, the next instruction is executed.

goto. Continues execution at a specified label.

recycle. Continues execution at the label on top of the *control* stack.

cpush. Pushes a label on the control stack.

Initialization of ARM is done by pushing the function symbols of the term to be normalized on the control stack (in pre-order, i.e. from left to right in a textual representation of the term).

The instructions described above are either available on common concrete machines (**goto** is always available, **recycle** corresponds to **return**, and **match** to **compare**) or can be implemented in a few instructions (**build** and **cpush**).

3.2.3 Simulating Pattern Matching with MTRSs

We now show how pattern matching of general lhs's can be simulated by MTRS rules, using the following example:

$$f(g(x), g(x)) \rightarrow r_1(x, x) \tag{3.1}$$

$$f(g(x), g(y)) \rightarrow r_2(x, y) \tag{3.2}$$

$$f(x, h) \rightarrow r_3(x) \tag{3.3}$$

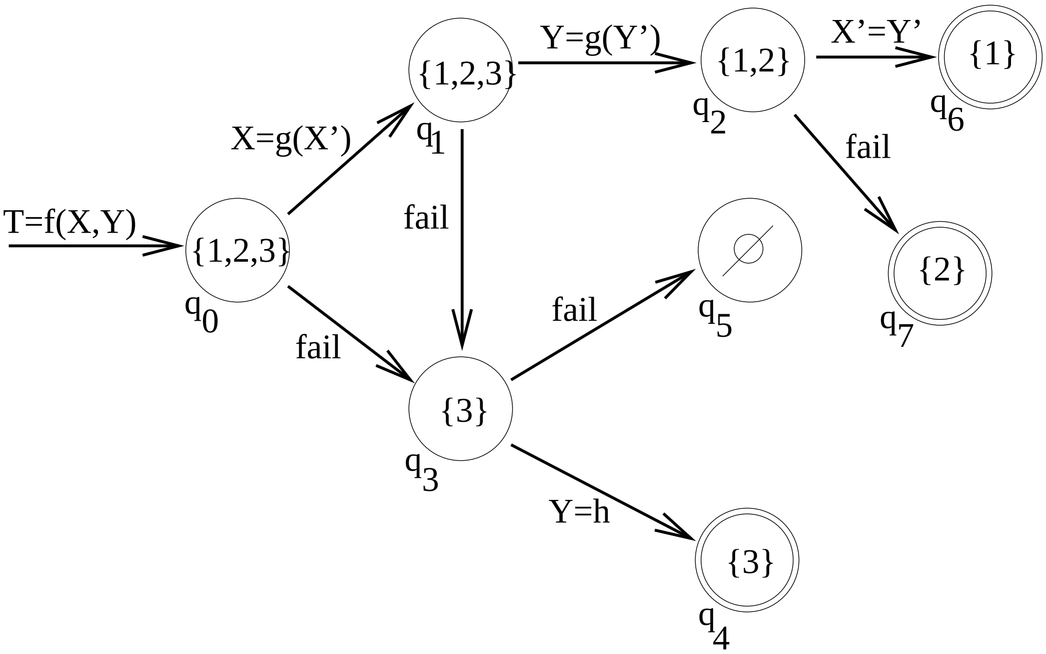


Figure 3.3: A tree matching automaton

This example contains overlapping rules and a nonlinearity, thus presenting the basic problems to be addressed by a TRS pattern-match compiler.

It is well-known that we can use tree matching automata [HO82, Wal91] for determining whether a given term T matches the lhs of one (or more) of a set of rewrite rules. In Figure 3.3, a matching automaton for this set of lhs's is depicted (syntactic specificity is used to disambiguate the overlapping patterns).

The states q_i of the automaton encode the set of patterns that might still match the term under consideration. Accepting states, in which it is known that T matches one or more rules, are indicated by a double circle. Based on the value of an argument position, there are success and failure transitions between states. It is understood that a failure transition is only made when no other transition is possible.

We will now show how this matching automaton is simulated by innermost rewriting with specificity of a TRS in which every rule has a minimal lhs.

There are three crucial ideas in this simulation. The first idea is that in innermost rewriting, the arguments of T are in normal form before a match with T is attempted, and when T fails to match, it is itself in normal form. Therefore, for every function symbol f, we introduce a *constructor variant* f_c which simulates f, that is $(\mathcal{S}(f_c) = f)$, and which implicitly indicates that matching has been attempted and failed. It follows that normal forms always consist entirely of constructor variants.

The second idea (found also in [Pet92]) is to encode the states of the automaton by auxiliary functions $q_0 \mapsto f$, $q_1 \mapsto f_g$, $q_2 \mapsto f_{gg}$, $q_3 \mapsto f_x$, $q_4 \mapsto r_3$, $q_5 \mapsto f_c$, $q_6 \mapsto r_1$ and $q_7 \mapsto r_2$, and the transitions by rules defining these functions (additional functions f_{ggx} and

f_{ggxy} are introduced to duplicate variables for the equality test). The map $\mathcal{S}$ is undefined on the auxiliary functions, i.e., f_g, f_{gg}, f_{ggx}, f_{ggxy}, f_{gge} and f_x.

The third idea is that failure transitions correspond to most general rules. Innermost rewriting with (syntactic) specificity ordering according to the TRS below exhibits the desired behaviour. For reference, we have annotated the rules with the state transitions they simulate.

$$h \longrightarrow h_c \tag{3.4}$$

$$g(x) \longrightarrow g_c(x) \tag{3.5}$$

$$f(g_c(x), y) \xrightarrow{q_0 \to q_1} f_g(x, y) \tag{3.6}$$

$$f(x, y) \xrightarrow{q_0 \to q_3} f_x(x, y) \tag{3.7}$$

$$f_g(x, g_c(y)) \xrightarrow{q_1 \to q_2} f_{gg}(x, y) \tag{3.8}$$

$$f_g(x, y) \xrightarrow{q_1 \to q_3} f_x(g_c(x), y) \tag{3.9}$$

$$f_{gg}(x, y) \xrightarrow{q_2 \to q_2'} f_{ggx}(x, x, y) \tag{3.10}$$

$$f_{ggx}(x', x, y) \xrightarrow{q_2' \to q_2''} f_{ggxy}(y, x', x, y) \tag{3.11}$$

$$f_{ggxy}(y', x', x, y) \xrightarrow{q_2'' \to q_2'''} f_{gge}(\mathrm{eq}(y', x'), x, y) \tag{3.12}$$

$$f_{gge}(\mathrm{true}, x, y) \xrightarrow{q_2''' \to q_6} r_1(x, y) \tag{3.13}$$

$$f_{gge}(\mathrm{false}, x, y) \xrightarrow{q_2''' \to q_7} r_2(x, y) \tag{3.14}$$

$$f_x(x, h_c) \xrightarrow{q_3 \to q_4} r_3(x) \tag{3.15}$$

$$f_x(x, y) \xrightarrow{q_3 \to q_5} f_c(x, y) \tag{3.16}$$

Note that in rule (3.9), a previously deconstructed term is reconstructed. At the cost of introducing extra variables, the cost of reconstruction can be avoided.

The function eq, which is used in rule 3.12 to test (syntactic) equality of its arguments, can easily be defined by a TRS if the signature is known and innermost rewriting with specificity ordering is assumed. We show an example definition of eq for the case that there is a constant f, a unary function g and a binary function h:

$$\mathrm{eq}(x, y) \rightarrow \mathrm{false} \tag{3.17}$$

$$\mathrm{eq}(f, f) \rightarrow \mathrm{true} \tag{3.18}$$

$$\mathrm{eq}(g(x), g(y)) \rightarrow \mathrm{eq}(x, y) \tag{3.19}$$

$$\mathrm{eq}(h(x_1, y_2), h(x_2, y_2)) \rightarrow \mathrm{and}(\mathrm{eq}(x_1, x_2), \mathrm{eq}(y_1, y_2)) \tag{3.20}$$

$$\mathrm{and}(x, y) \rightarrow \mathrm{false} \tag{3.21}$$

$$\mathrm{and}(\mathrm{true}, x) \rightarrow x \tag{3.22}$$

3.3 Minimal Term Rewriting Systems

We first define the forms that *minimal rules* may assume:

Definition 22 (Minimal Rule) *Let* $\Re = \langle \Sigma, R \rangle$ *be a TRS, and* $r : s \to t$ *a rule in* R. *The rule* r *is called* minimal *if it is left-linear and it has one of the following six forms:*

$$\begin{array}{rrcll}
\mathbf{C}: & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, g(\vec{y}), \vec{z}) & \\
\mathbf{R}: & f(y) & \to & y & \\
\mathbf{M}: & f(\vec{x}, g(\vec{y}), \vec{z}) & \to & h(\vec{x}, \vec{y}, \vec{z}) & \\
\mathbf{A}: & f(\vec{x}, \vec{z}) & \to & h(\vec{x}, y, \vec{z}) & (y \textit{ is } x_i \textit{ or } z_i) \\
\mathbf{D}: & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, \vec{z}) & (\mid \vec{y} \mid \neq 0) \\
\mathbf{I}: & f(\vec{x}) & \to & h(\vec{x}) &
\end{array}$$

We have labeled the forms with mnemonics reminding of their basic purpose (in the context of innermost rewriting). The mnemonic **C** stands for *continuation*, in the sense that h is the continuation after the evaluation of g. Conversely, **R** stands for *return*, in the sense that control is passed to a continuation if that was issued earlier, or rewriting is finished if there is no such continuation. Rules of the form **M** take apart a term, when there is a *match* of the symbol g. The forms **A**, **D** and **I** are for *addition, deletion* and *identity* on the set of variables.

Definition 23 (Minimal Term Rewriting System) *A TRS* $\Re$ *is called a Minimal Term Rewriting System (MTRS) if all its rules are minimal, and the following conditions hold:*

1. *Every function symbol is either fully defined or a free constructor.*
2. *For all rules* $l \to r$ *in* $\Re$, *ofs*(r) *is a defined function.*
3. *All* **M***-rules defining a function* f *are mutually exclusive.*

Constraint 1 ensures that normal forms consist entirely of free constructors.

Constraint 2 simplifies the interpretation of MTRS rules as machine code; due to this constraint, the *ofs* of a rhs can always be interpreted as an address where execution of the machine program should continue.

Constraint 3 ensures that the rewrite relation of an MTRS, restricted to innermost rewriting with syntactic specificity ordering is deterministic when there is exactly one most general rule for every defined function symbol. This follows from the fact that all forms except **M** are unconditional.

3.4 The Abstract Rewriting Machine

The rules of MTRSs can be viewed as (short sequences of) instructions for an abstract machine with three stacks C (control), A (arguments) and T (traversal), a heap H, a program counter p and a program P, visualized as a tuple $\langle p, P, C, T, A, H \rangle$. In Figure 3.4, we give an algebraic specification of this machine, which we will now explain in text.

$$
\begin{array}{rcl}
\langle \mathbf{goto}(l)\cdot p, P, C, T, A, H\rangle & = & \langle \mathrm{get}(l,P), P, C, T, A, H\rangle \\
\langle \mathbf{cpush}(l)\cdot p, P, C, T, A, H\rangle & = & \langle p, P, l\cdot C, T, A, H\rangle \\
\langle \mathbf{recycle}\cdot p, P, l\cdot C, T, A, H\rangle & = & \langle \mathrm{get}(l,P), P, C, T, A, H\rangle \\
\langle \mathbf{recycle}\cdot p, P, \epsilon, \epsilon, a\cdot\epsilon, \cdots a : t \cdots\rangle & = & t \\
\langle \mathbf{build}(c,n)\cdot p, P, C, T, a_1\cdots a_n\cdot A, H\rangle & = & \langle p, P, C, T, a\cdot A, a : c(a_1,\ldots,a_n)\cdot H\rangle \\
 & & \text{where } a \text{ is fresh} \\
\langle \mathbf{match}(c,n,l)\cdot p, P, C, T, a\cdot A, H\rangle & = & \langle \mathrm{get}(l,P), P, C, T, a_1\cdots a_n\cdot A, H\rangle \\
 & & \text{when } H = \cdots a : c(a_1,\ldots,a_n)\cdots \\
\langle \mathbf{match}(c,n,l)\cdot p, P, C, T, a\cdot A, H\rangle & = & \langle p, P, C, T, a\cdot A, H\rangle \\
 & & \text{otherwise} \\
\langle \mathbf{tpusha}(n)\cdot p, P, C, T, A, H\rangle & = & \langle p, P, C, T, \mathrm{top}(n,T)\cdot A, H\rangle \\
\langle \mathbf{apusha}(n)\cdot p, P, C, T, A, H\rangle & = & \langle p, P, C, T, \mathrm{top}(n,A)\cdot A, H\rangle \\
\langle \mathbf{tdrop}(n)\cdot p, P, C, a_1\cdots a_n\cdot T, A, H\rangle & = & \langle p, P, C, T, A, H\rangle \\
\langle \mathbf{skip}(n)\cdot p, P, C, T, a_1\cdots a_n\cdot A, H\rangle & = & \langle \mathbf{skip}(n)\cdot p, P, C, a_n\cdots a_1\cdot T, A, H\rangle \\
\langle \mathbf{retract}(n)\cdot p, P, C, a_1\cdots a_n\cdot T, A, H\rangle & = & \langle \mathbf{retract}(n)\cdot p, P, C, T, a_n\cdots a_1\cdot A, H\rangle \\
\mathrm{top}(0, a\cdot T) & = & a \\
\mathrm{top}(\mathrm{s}(n), a\cdot T) & = & \mathrm{top}(n, T) \\
\mathrm{get}(l, l : S\cdot P) & = & S \\
\mathrm{get}(l, l' : S\cdot P) & = & \mathrm{get}(l, P) \\
 & & \text{when } l \neq l'
\end{array}
$$

Figure 3.4: An algebraic specification of ARM instructions.

The program counter p denotes the fragment of the program P which is currently being executed. The stacks contain references to terms in the heap. In the heap, $a : t$ means that the term t has address a.

The **goto** instruction replaces the current fragment by a fragment of P, which is obtained as $\mathrm{get}(l, P)$, where l is a unique label identifying the fragment.

The **cpush** instruction pushes a label onto the C (mnemonic for *control*) stack, whence it may be removed by a **recycle** instruction, which causes execution to continue at the label obtained from C.

A **build** instruction builds a term in the heap from a function symbol and a number of (references to) terms from the A (mnemonic for *argument*) stack.

The **match** instruction matches the term referred to by the address on top of the A stack against a certain function symbol. On success, it decomposes this term, leaving references to its arguments on the A stack, and continues at a specified label. On failure, the next instruction of the current fragment is executed. Strictly speaking, the arity argument of **match** is redundant, because the number of arguments may be obtained by inspecting the term on top of the A stack.

The instruction **tpusha** pushes an argument from the T (mnemonic for *traversal*) stack onto the A stack, the **apusha** instruction pushes an argument from the A stack onto the A stack, the **skip** instruction moves a number of terms from the A stack to the T stack, **retract** does this in the reverse direction, and the **tdrop** instruction removes a number of terms from the T stack. We assume the programs to be such that top is never applied to an empty stack.

As shown, the rule for **tdrop** is actually short for two rules, $\langle \mathbf{tdrop}(0) \cdot p, P, C, T, A\rangle = \langle p, P, C, T, A\rangle$ and $\langle \mathbf{tdrop}(\mathrm{s}(n)) \cdot p, P, C, t \cdot T, A\rangle = \langle \mathbf{tdrop}(n) \cdot p, P, C, T, A\rangle$ when $n \neq 0$. We have used the same abbreviation mechanism for the specification of **build**, **skip** and **retract**.

All instructions of the ARM machine can be implemented efficiently on modern microprocessors. Usually, **goto** and **recycle** are available as native instructions, **cpush**, **match**, **apusha**, **tpusha**, and **tdrop** can be implemented in one or a few native instructions, and **build**, **skip**, and **retract** can be implemented in kn instructions, where k is a small factor, and n is the parameter of the instruction.

Furthermore, only the implementation of **build** requires write-access to heap storage, and only **match** requires read-access to such storage. The other instructions only access stack storage, which is important when considering modern caching techniques.

Finally, it is clear that large sequences of **match** instructions (of length larger than 3) should be implemented table driven rather than iterative.

We believe that this set is the minimal set for which efficiency can be preserved in the translation from general TRSs. In [HF$^+$96], concrete execution times are reported concerning an implementation based on ARM technology.

$$\begin{array}{rrcll}
\mathbf{C}: & f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},g(\vec{y}),\vec{z}) & \begin{cases} (g \text{ defined}) & \mathbf{f}:\mathbf{cpush}(h);\mathbf{goto}(g) \\ (g \text{ free}) & \mathbf{f}:\mathbf{build}(g,|\,\vec{y}\,|);\mathbf{goto}(h) \end{cases} \\
\mathbf{M}: & f^{|\vec{x}|}(\vec{x},c(\vec{y}),\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \mathbf{f}:\mathbf{match}(c,h) \\
\mathbf{A}: & f^{|\vec{x}|}(\vec{x},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},x_k,\vec{z}) & \mathbf{f}:\mathbf{tpusha}(|\,\vec{x}\,|-k);\mathbf{goto}(h) \\
\mathbf{A}: & f^{|\vec{x}|}(\vec{x},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},z_k,\vec{z}) & \mathbf{f}:\mathbf{apusha}(k-1);\mathbf{goto}(h) \\
\mathbf{D}: & f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},\vec{z}) & \mathbf{f}:\mathbf{tdrop}(|\,\vec{y}\,|);\mathbf{goto}(h) \\
\mathbf{I}: & f^{n}(\vec{x}) & \rightarrow & h^{m}(\vec{x}) & \begin{cases} (m \geq n) & \mathbf{f}:\mathbf{skip}(m-n);\mathbf{goto}(h) \\ (n > m) & \mathbf{f}:\mathbf{retract}(n-m);\mathbf{goto}(h) \end{cases} \\
\mathbf{R}: & f^{1}(x) & \rightarrow & x & \mathbf{f}:\mathbf{recycle}
\end{array}$$

Figure 3.5: The instruction mapping

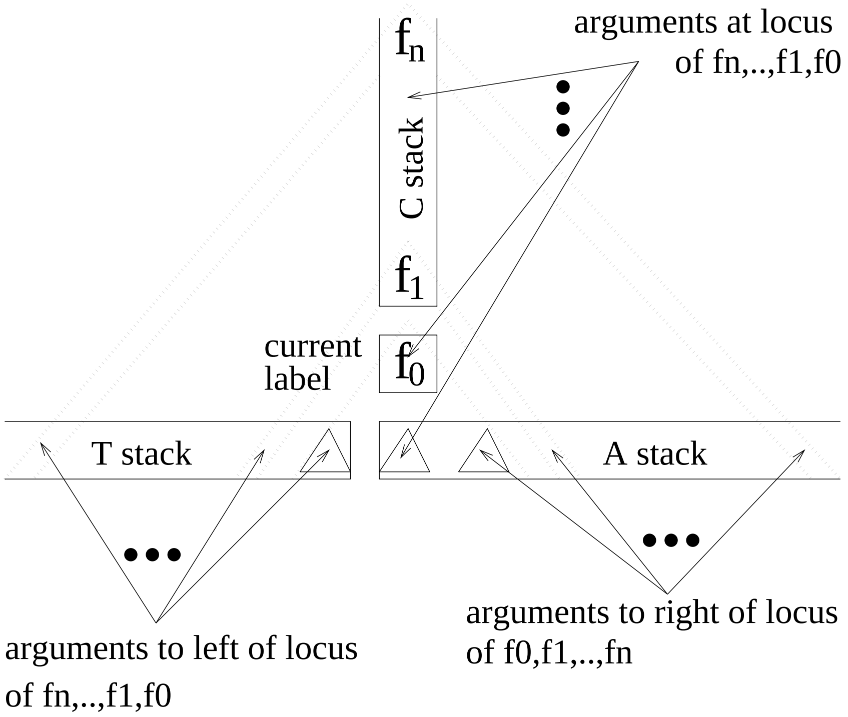


Figure 3.6: The invariant maintained by the mapping in Figure 3.5

3.5 Interpreting MTRSs as ARM programs

The actual interpretation of MTRS rules as instructions for ARM is based on the following invariant. When any MTRS rule[1] $f(\vec{x}, \vec{t}) \rightarrow g(\vec{x}, \vec{t'})$ applies, the terms corresponding to $\vec{x}$ are on the T stack, and the terms corresponding to $\vec{t}$ are on the A stack. In the transformation of Section 3.7, the original function symbols have all arguments on the argument stack; **I** rules move them gradually to the traversal stack. Newly introduced function symbols are annotated with the number of arguments that are already on the traversal stack. We will call this number the *locus* of a function symbol. A conflict regarding a *locus* can always be solved by introducing an **I** rule, which is the reason we will ignore *loci* from Section 3.6 till the end of this chapter. In the concrete compiler described in Chapter 5, the insertion of **I** rules is dealt with explicitly.

In Fig. 3.5, we show the mapping of MTRS forms onto instruction sequences, and in Fig. 3.6 the invariant maintained by this mapping.

First, we view defined symbols as labels in the machine program. A rule with f as outermost function symbol on the lhs *defines* the instructions at label f, and a rule with h as outermost function symbol on the rhs *uses* the label h, i.e. it causes execution to continue at label h. (for rules with the same label, we simply concatenate the code, taking care that the code for a general rule is put at the end).

Form **C** has two labels on the rhs, of which the innermost label g is interpreted as the label where execution should continue, whereas the outermost label h is pushed on the C stack for future reference.

Form **R** has no labels on the rhs, which is taken to mean that execution should continue at a label popped from the C stack.

Second, the similarity of the variable configurations on lhs and rhs of an MTRS rule is exploited by representing the term to be rewritten as follows (see Fig. 3.6).

The top-symbol f_n of the entire term being rewritten, is at the bottom of the C stack (the C stack is shown upside-down), all arguments left of the locus of f_n are on the T stack, all arguments right of the locus of f_n are on the A stack, and, recursively, the argument at the locus (with top-symbol f_{n-1}) is represented less deep than f_n on the C, T and A stacks.

The symbol f_0 is the current label, which does not reside on the C stack, but is expressed in the current state of the machine (i.e., the program counter p in the tuple $\langle p, P, C, T, A \rangle$).

Initially, this invariant is satisfied by traversing the input term in pre-order, pushing all encountered function symbols on the C stack (consistent with the fact that the function symbols of the input term have locus 0), and starting the machine with the **recycle** instruction.

It can be verified that the mapping from MTRS rules to instructions given in Fig. 3.5 implements rightmost innermost rewriting, by checking that the machine instructions associated with a particular MTRS rule satisfy the invariant.

---

[1] An **R** rule does not conform to the format that follows. For **R** rules we take $\mid \vec{x} \mid = 0$.

For example, after the replacement of most **goto** instructions by the code at their destination label, the program generated from the example in Figure 3.2 becomes:

$$\begin{array}{ll}
zero: & \mathbf{build}(zero_c, 0); \mathbf{recycle} \\
succ: & \mathbf{build}(succ_c, 1); \mathbf{recycle} \\
plus: & \mathbf{match}(zero_c, plus\_zero); \\
 & \mathbf{match}(succ_c, plus\_succ); \\
 & \mathbf{build}(plus_c, 2); \mathbf{recycle}; \\
plus\_zero: & \mathbf{recycle}; \\
plus\_succ: & \mathbf{cpush}(succ); \mathbf{goto}(plus);
\end{array}$$

It is readily verified that the state $\langle \mathbf{recycle}, P, zero \cdot succ \cdot zero \cdot succ \cdot plus \cdot \epsilon, \epsilon, \epsilon \rangle$, with P the program above, is provably equal to $\langle \epsilon, P, \epsilon, \epsilon, succ_c(succ_c(zero_c)) \cdot \epsilon$ in the algebraic semantics given in Fig. 3.4.

3.6 Term Rewriting Simulations

In this section, we define the notion of *simulation* of a TRS by another TRS. Our notion of simulation is similar to the notions of *refinement* and *(bi)simulation*, which are used in Concurrency Theory and Automata Theory [LV95]. In contrast to [KW95b, KW96], and with thanks to Bas Luttik, Wan Fokkink and Jaco van de Pol (see [FvdP96]), we will proceed as follows. First, we define simulation on *abstract rewrite systems*, and then, we define a specialization to TRSs, which is the notion of simulation we will use in the rest of the chapter.

A simulation of an ARS (A, R) by an ARS (B, S) is characterized by two mappings $\phi : B \rightarrow A$ and $\psi : A \rightarrow B$. The intuition for the mapping ϕ, which in general is only partially defined, is that the reduction tree of $a \in A$ with respect to R is mimicked by the reduction tree of each $b \in \phi^{-1}(a)$ with respect to S. The mapping ψ selects for each $a \in A$ an interpretation in $\phi^{-1}(a)$, so in particular $\phi(\psi(a)) = a$.

Definition 24 *A* simulation *of an ARS (A, R) by an ARS (B, S) consists of two mappings:*

1. *a partially defined mapping $\phi : B \rightarrow A$;*
2. *a mapping $\psi : A \rightarrow B$ such that $\phi(\psi(a)) = a$ for each $a \in A$.*

The second condition implies that ϕ is surjective and that ψ is injective. However, typically ϕ is *not* injective (i.e., the simulating system may contain multiple representations of the same element in the simulated system) and ψ is *not* surjective (i.e., the simulating system contains 'administrative junk'). In our case, A will be a (proper) subset of B, and ψ is the identity.

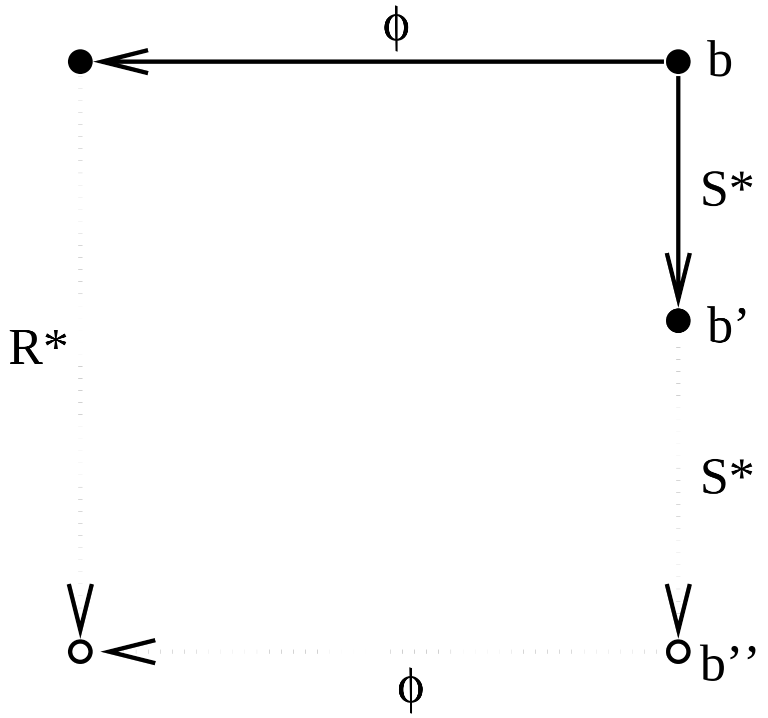


Figure 3.7: Soundness.

3.6.1 Soundness and Completeness

In this and the following three sections we assume as general notation that the ARS (A, R) is simulated by the ARS (B, S) by means of the mappings $\phi : B \to A$ and $\psi : A \to B$. Suppose that $\phi(b)$ is defined for some $b \in B$. Soundness of the simulation means that each finite S-reduction of b is a mimicking of some finite R-reduction of $\phi(b)$. In Figure 3.7, the definition of Soundness is illustrated. In this figure, existentially quantified elements are represented as dashed arrows and open circles, whereas universally quantified elements are represented as solid arrows and filled circles.

Definition 25 *(Soundness) A simulation is* sound *if for each $b, b' \in B$ with $\phi(b)$ defined and bS^*b', there is a term $b'' \in B$ with $b'S^*b''$ and $\phi(b'')$ defined and $\phi(b)R^*\phi(b'')$.*

Conversely, completeness means that each R-step from $\phi(b)$ can be mimicked by a finite S-reduction of b with length greater than zero. In Figure 3.8, the definition of Completeness is illustrated.

Definition 26 *(Completeness) A simulation is* complete *if for each $a \in A$ and $b \in B$ with $\phi(b)$ defined and $\phi(b)Ra$, there is a term $b' \in B$ with bS^+b' and $\phi(b')$ is defined and $\phi(b') = a$.*

A weaker version of completeness helps to ensure that if there exists an R-step from $\phi(b)$, then at least one of these R-steps can be mimicked by a finite S-reduction of b with length greater than zero.

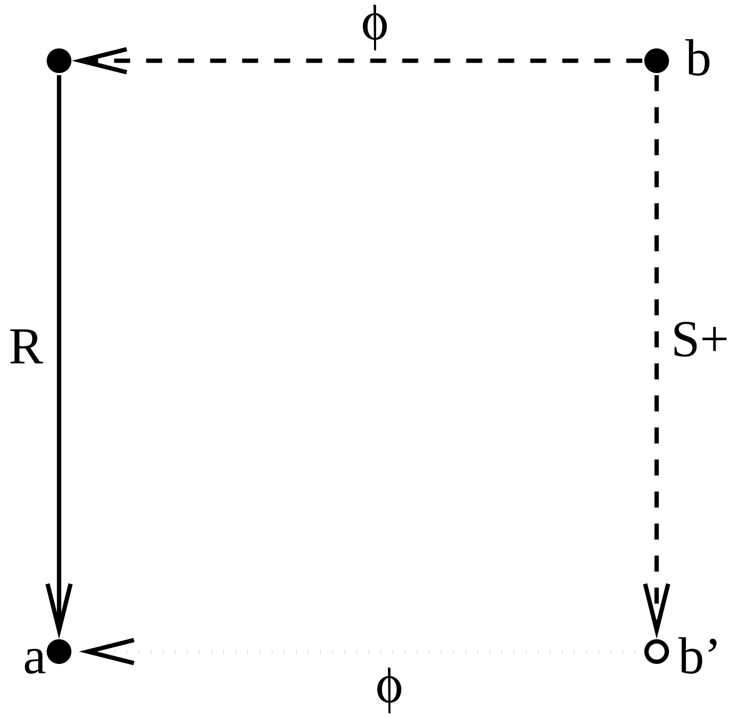


Figure 3.8: Completeness.

Definition 27 *(Weak Completeness) A simulation is* weakly complete *if for each $b \in B$ with $\phi(b)$ defined and b a normal form for S, $\phi(b)$ is a normal form for R.*

It is easily seen that the composition of two simulations is again a simulation. Moreover, soundness, completeness and weak completeness are preserved under composition.

3.6.2 Termination Conservation

A simulating system should terminate whenever the simulated system terminates. Total conservation (Definition 28) ensures that termination properties are preserved with respect to the mapping ϕ, while conservation (Definition 29) only ensures that termination properties are preserved with respect to the mapping ψ.

We will use the convention that sentences which contain occurrences of the expression '(weak)' or '(weakly)' can be read both with and without the word 'weak' or 'weakly' at those places, respectively.

Definition 28 *(Total Conservation of (weak) termination) A simulation* totally conserves (weak) termination *if for each $a \in A$ for which R is (weakly) terminating, also S is (weakly) terminating for each $b \in \phi^{-1}(a)$.*

Definition 29 *(Conservation of termination) A simulation* conserves termination *if for each $a \in A$ for which R is terminating, also S is terminating for $\psi(a)$.*

Proposition 3 *The following implications hold:*

1. *total conservation of (weak) termination* $\Rightarrow$ *conservation of (weak) termination;*
2. *total conservation of termination + completeness* $\Rightarrow$ *total conservation of weak termination.*

Proof We assume that the ARS (B, S) simulates the ARS (A, R) by means of the pair (ϕ, ψ).

1. If the simulation totally conserves (weak) termination, then it also conserves (weak) termination, simply because $\psi(a) \in \phi^{-1}(a)$ for each $a \in A$.

2. Suppose that the simulation (ϕ, ψ) totally conserves termination and is complete. Let R be weakly terminating for $a \in A$, and let $b \in \phi^{-1}(a)$. We show that S is weakly terminating for b, by induction on the length of the shortest reduction to a normal form of a.

   If a is a normal form for R, then total conservation of termination yields that S is terminating for $b \in \phi^{-1}$, so because termination implies weak termination, S is also weakly terminating for b.

   Next, suppose that we have proven the case for reductions to normal form of length n, and let the shortest reduction to a normal form of a have length $n+1$. Then there exists a reduction aRa' where the shortest reduction to a normal of a' has length n. Since $\phi(b) = a$, completeness yields that $bS^{+}b'$ for some $b' \in B$ with $\phi(b') = a'$. Since R is weakly terminating for a' with a shortest normalization reduction of length n, induction yields that S is weakly terminating for b'. Since bSb', it follows that S is also weakly terminating for b.

■

3.6.3 Reachabilitty

Usually, the properties above are too strong, and we must restrict to the *reachable* part, as noted first by [Ver95]. Reachability is defined as follows:

Definition 30 *(Reachability)* $b \in B$ *is* reachable *if* $\phi(a)S^{*}b$ *for some* $a \in A$.

Lemma 1 *Let* (ϕ, ψ) *be a simulation of* (A, R) *by* (B, S)*, and let* $\overline{\phi}$ *denote the restriction of* ϕ *to the reachable part of* B*. Then* $(\overline{\phi}, \psi)$ *is also a simulation of* (A, R) *by* (B, S)*. Furthermore, if* (ϕ, ψ) *satisfies soundness or completeness or conservation of termination, then* $(\overline{\phi}, \psi)$ *also satisfies this property.*

Proof Clearly $\psi(a)$ is reachable for each $a \in A$. Hence, $\overline{\phi}$ is defined for each $\psi(a)$, and $\overline{\phi}(\psi(a)) = a$. So $(\overline{\phi}, \psi)$ is a simulation.

Assume that (ϕ, ψ) is sound; we show that $(\overline{\phi}, \psi)$ is also sound. Let $\overline{\phi}(b)$ be defined, and $bS^{*}b'$. Then soundness of (ϕ, ψ) yields that $b'S^{*}b''$ where $\phi(b'')$ is defined and $\phi(b)R^{*}\phi(b'')$.

Since b is reachable and $bS^*b'S^*b''$, it follows that b'' is also reachable. Hence, $\overline{\phi}$ is defined for b'', and $\overline{\phi}(b)R^*\overline{\phi}(b'')$. So $(\overline{\phi}, \psi)$ is sound.

Assume that (ϕ, ψ) is complete; we show that $(\overline{\phi}, \psi)$ is also complete. Let $\overline{\phi}(b)$ be defined, and $\overline{\phi}(b)Ra$. Then completeness of (ϕ, ψ) yields that bS^+b' where $\phi(b') = a$. Since b is reachable and bS^+b', it follows that b' is also reachable. Hence, $\overline{\phi}$ is defined for b', and $\overline{\phi}(b') = a$. So $(\overline{\phi}, \psi)$ is complete.

Finally, if (ϕ, ψ) conserves termination, then the same holds for $(\overline{\phi}, \psi)$, simply because this property does not depend on ϕ, but on ψ ∎

In the rest of this chapter, we will always assume a restriction to the reachable part of B.

3.6.4 A Simulation Map for Term Rewriting

We now turn to term rewriting. We will assume the sets A and B to be $T(\Sigma)$ and $T(\Sigma')$, respectively, where $\Sigma = (\mathcal{F}, \mathcal{V})$ and $\Sigma' = (\mathcal{F}', \mathcal{V})$, with $\mathcal{F} \subseteq \mathcal{F}'$, so $A \subseteq B$ as indicated in the previous section. Similarly, we take ψ to be the identity on $T(\Sigma)$.

We will impose some additional structure on ϕ, requiring that it is the homomorphic extension of a partial map that respects arity $\mathcal{S} : \mathcal{F}' \rightharpoonup \mathcal{F}$, which we call simulation map. Furthermore, we require that ϕ is the identity on variables.

The fact that (ϕ, ψ) is a simulation and ψ is the identity imply that $\forall f \in \mathcal{F} : \mathcal{S}(f) = f$. We will call a pair (ϕ, ψ) such that ϕ is the homomorphic extension of $\mathcal{S}$ that is the identity on variables, and ψ is the identity on $T(\Sigma)$, the simulation *induced* by the simulation map $\mathcal{S}$.

In the rest of this chapter, we will denote the identity on $\mathcal{F}$ by $\mathrm{I}_{\mathcal{F}}$, and we let $\mathcal{D}_{\mathcal{S}}$ be a predicate that holds precisely for all terms in $T(\Sigma')$ for which ϕ is defined.

As an example, consider $\mathcal{F} = \{f, a\}$ and $\mathcal{F}' = \{f, a, f_c, h\}$. In this example, f_c is a variant (a so-called constructor variant, discussed further in the sequel) of f with $\mathcal{S}(f_c) = f$, and h is an auxiliary function that has no counterpart in $\mathcal{F}$. Supposing that the arity of f is 1, and the arity of a is 0, we have (by partial homomorphic extension) that $\mathcal{S}(f(f_c(a))) = f(f(a))$, and $\neg\mathcal{D}_{\mathcal{S}}(f(h(a)))$, so $\mathcal{S}(f(h(a)))$ is undefined.

3.6.5 Simple Simulation

In general, it is a very hard task to prove simulation of one TRS by another. Fortunately, we do not need to prove arbitrary simulations. In this chapter, we will construct simulating TRS in a series of small transformation steps, in such a way that the steps can be proven correct one after another. Usually, a transformation step replaces a single rule by two new rules, such that when the two rules are applied after each other, the original rule is simulated. The general pattern is to replace a rule of the form

$$l \to r$$

by two rules of the form

$$l \to f(\vec{t})$$

$$f(\vec{x}) \to r[\vec{t} \;/\; \vec{x}]$$

Where the symbol f does not occur in the original TRS, and the intentionally vague notation $[\vec{t} \;/\; \vec{x}]$ should mean 'a replacement of terms from $\vec{t}$ by variables from $\vec{x}$'. From the form of the second rule, it is clear that f never occurs in normal forms, so we can safely take the identity as simulation map (leaving the simulation map undefined on f).

We will call a useful class of such transformations *simple simulations.* Below, we provide a lemma that can be used to prove simple simulations.

Lemma 2 *(Simple Simulation) Let* $\Re = \langle \Sigma, R \rangle$ *and* $\Re' = \langle \Sigma', R' \rangle$*, then* $\mathrm{I}_{\mathcal{F}}$ *induces a simulation of* $\Re$ *by* $\Re'$*, if:*

1. $\Sigma' = \Sigma \cup \{f\}$ $(f \notin \Sigma)$;
2. $R = R_0 \cup \{r_0 : l \to r\}$ $(r_0 \notin R_0)$;
3. $R' = R_0 \cup \{r_1 : l \to f(\vec{t}), r_2 : f(\vec{x}) \to r'\}$;
4. $s \to_{r_0} t \wedge s \to_{r_1} s' \Leftrightarrow s \to_{r_1} s' \to_{r_2} t$;
5. *All (sub)terms occurring in* $\vec{t}$ *also occur in* l *or in* r.

Proof We have to prove completeness, soundness, and termination conservation.

Completeness is trivial, it follows directly from requirement 4.

For soundness, we must prove that for each $s, t \in T(\Sigma')$ with ϕ defined and sR'^*t, there is a u with tR'^*u and $\phi(u)$ defined and $\phi(s)R^*\phi(u)$. First, we observe that reductions according to r_2 are only possible on terms created by applications of r_1. Because r_2 is most general, we can construct a term u with $\phi(u)$ defined by rewriting all r_2-redexes in t. Because r_2 has no overlap with other rules, we can replace all r_1-reductions in $sR'^*tR'^*u$ by r_0-reductions, delete the application of r_2, and obtain sR^*u, which completes the proof because ϕ is the identity on $T(\Sigma)$.

We prove termination conservation by considering the number of r_1-contractions. If there are no r_1-contractions in an infinite R'-sequence starting in a term t with ϕ defined, there are no r_2-contractions either, so the infinite sequence is itself an R-sequence. If there is only a finite number of r_1-contractions in an infinite sequence, there can only be an infinite number of r_2-contractions if there is some context $C[]$ in which (descendants of) an r_2 redex can be duplicated infinitely many times. But because r_2 has no overlap with other rules, this means that (descendants of) the r_1-redex can already be duplicated infinitely many times in $C[]$, which is a contradiction, so all r_1 and r_2-contractions occur in a finite prefix of the infinite sequence, and the infinite suffix corresponds to an infinite R-sequence. Finally, if there is an infinite number of r_1-contractions, then there is also an infinite number of r_0-contractions possible, because all subterms in an instantiated rhs of r_1 are also in an instantiated rhs of r_0, and an instance of the rhs of r_1 itself can only be contracted by r_2, with the same result as a direct contraction by r_0 ■

3.7 Simulating left-linear TRSs by MTRSs

We will now show that every left-linear TRS can be simulated by an MTRS, under the assumption of innermost rewriting with specificity ordering. Actually, we first provide a terminating transformation that transforms any *simply complete* left-linear TRS into a simulating MTRS, regardless of the reduction strategy. This first transformation consists of two parts. In Section 3.7.1, we remove complicated pattern matching to such an extent, that all remaining pattern matching is done by MTRS rules, and in Section 3.7.2, we replace remaining non-MTRS rules by simulating MTRS rules.

Then, in Section 3.7.3 a simulation of left-linear TRSs by simply complete TRSs is given. Only this last simulation is dependent on innermost rewriting and specificity ordering.

In general, the transformations are composed of many steps. Because simulation is transitive, it suffices to prove simulation for every individual step.

3.7.1 Transforming complicated lhs's

The pattern matching transformation is done by the function **sim**. We give here a specification of **sim**, and prove that applying **sim** to a simply complete TRS $\langle \Sigma, R\rangle$ yields a TRS $\langle \Sigma', R'\rangle$ such that the simulation induced by $\mathrm{I}_{\mathcal{F}}$ is a simulation of R by R', and the lhs of all non-minimal rules in R' is most general. The specification of **sim** is itself nonambiguous and terminating, so it can be used as a pattern matching compiler.

In the specification, we will extensively use *union* of TRSs:

$$\langle \Sigma, R\rangle \cup \langle \Sigma', R'\rangle \quad = \quad \langle \Sigma \cup \Sigma', R \cup R'\rangle$$

Given some index set $I = \{i_1, \ldots, i_n\}$, we will use the notation $\bigcup_{i \in I} T_i$ for the (finite) union $T_{i_1} \cup \ldots \cup T_{i_n}$.

If all non-minimal rules in R are *most general*, then:

$$[\text{sim-base}]\ \mathbf{sim}\langle \Sigma, R\rangle \quad = \quad \langle \Sigma, R\rangle.$$

Otherwise, let i be the least index such that for some rule $f(\vec{t}\,) \to s \in R$, we have $t_i \notin V$, and let $G = \{g | f(\vec{t}\,) \to r \in R \wedge t_i = g(\vec{u})\}$, the set of function symbols found at position i of lhs's defining f in R. Then, taking $|\ \vec{x}\ | = |\ \vec{t}\ | = i - 1$, and $f^S, f_g \notin \Sigma$ fresh symbols, we have:

$$[\text{sim-rec}]\ \mathbf{sim}\langle \Sigma, R\rangle \quad = \quad (\textstyle\bigcup_{g \in G} \mathit{Match}_g \cup \mathit{Matched}_g) \cup \mathit{Skip} \cup \mathit{Other},$$

where

$$\begin{array}{rcl}
\mathit{Match}_g & = & \langle \{f, g, f_g\}, \{(m1 :) f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z})\}\rangle; \\
\mathit{Matched}_g & = & \mathbf{sim}\langle \Sigma \cup \{f_g, f^S\},\ \{(m2 :) f_g(\vec{t}, \vec{u}, \vec{v}) \to r |\ (m3 :) f(\vec{t}, g(\vec{u}), \vec{v}) \to r \in R\} \\
 & & \qquad \cup \{(m4 :) f_g(\vec{x}, \vec{y}, \vec{z}) \to f^S(\vec{x}, g(\vec{y}), \vec{z})\}\rangle; \\
\mathit{Skip} & = & \langle \Sigma \cup \{f^S\}, \{(s1 :) f(\vec{x}) \to f^S(\vec{x})\}\rangle; \\
\mathit{Other} & = & \mathbf{sim}\langle \Sigma \cup f^S,\ \{(o1 :) f^S(\vec{t}\,) \to s |\ (o2 :) f(\vec{t}\,) \to s \in R \wedge\ \vec{t}_i \in V\} \\
 & & \qquad \cup \{(o3 :) r |\ r \in R \wedge \mathit{ofs}(\mathit{lhs}(r)) \neq f\}\rangle.
\end{array}$$

An intuitive explanation of [sim-rec] is, that $Match_g$ has a rule $m1$ that matches a symbol g at position i, $Matched_g$ deals with a succesful match of g at position i, by either completing a match of $m3$ by applying $m2$, or restoring the lhs of $m1$ (up to f^S) by applying $m4$, *Skip* just replaces f by f^S (with the effect of sharing an 'automaton' state between reconstructed terms and terms for which matching fails right away), and *Other* simulates the rules $o2$ that have a variable at position i with rules $o1$, and rules $o3$ for other function symbols than f.

Let $NVP(R)$ be the number of paths to nonvariable proper subterms of lhs's of the rewrite rules R. It is clear that $NVP(R)$ is a well-founded measure on TRSs. It is easily established that, when read from left to right, the recursive rule [sim-rec] is decreasing in this measure. Furthermore, the conditions are decidable, so the specification of **sim** is an executable specification that can be used as a pattern-match compiler.

Theorem 1 *Let* $\langle \Sigma', R' \rangle = \mathbf{sim}(\langle \Sigma, R \rangle)$. *Then* $\mathrm{I}_{\mathcal{F}}$ *induces a simulation of* R *by* R'.
Proof We need to prove completeness, soundess and termination conservation.

Completeness By induction on $NVP(R)$. If $NVP(R) = 0$, we have case [sim-base], which is trivially complete. Otherwise, [sim-rec] must be applied. Because the simulation map is I_Σ, we only have to consider terms in $T(\Sigma)$. There are three cases: either a rule of type $m3$ is applied, or another rule defining f, or a rule defining another function symbol than f. A rule of type $m3$ is simulated by applying $m1$ and then $m2$; other rules defining f are simulated by applying $s1$ and then $o1$; rules defining other symbols than f are available as rules of type $o3$

Soundness With induction on $NVP(R)$. When $NVP(R) = 0$, it is clear we have case [sim-base], and soundness is trivial. Otherwise, let i be the least index such that there is a rule $r \in R$ with $lhs(r)|_i \notin V$. Without loss of generality, we can assume r to be $f(\vec{t}, g(\vec{u}), \vec{v}) \to s$, with $|\ \vec{t}\ | = i - 1$. We have to prove that

$$\forall st\ (\mathcal{D}_{\mathcal{S}}(s) \wedge sR'^{*}t) \Longrightarrow \exists u \mathcal{D}_{\mathcal{S}}(u) \wedge tR'^{*}u \wedge \mathcal{S}(s)R^{*}\mathcal{S}(t)$$

By the induction hypothesis, we may assume $Matched_g$ to simulate

$$\begin{array}{l}\langle \Sigma \cup \{f_g, f^S\}, \\ \{(m2:) f_g(\vec{t}, \vec{u}, \vec{v}) \to r|\ (m3:) f(\vec{t}, g(\vec{u}), \vec{v}) \to r \in R\} \\ \cup\{(m4:) f_g(\vec{x}, \vec{y}, \vec{z}) \to f^S(\vec{x}, g(\vec{y}), \vec{z})\}\rangle\end{array},$$

and *Other* to simulate

$$\begin{array}{l}\langle \Sigma \cup f^S, \\ \{(o1:) r^S|\ r \in R \wedge ofs(lhs(r)) = f \wedge lhs(r)|_i \in V\} \\ \cup\{(o3:) r|\ r \in R \wedge ofs(lhs(r)) \neq f\}\rangle\end{array}.$$

It is clear that when the sequence $sR'^{*}t$ does not contain a (sub)term with a function symbol f_g or f^S, $sR^{*}t$ holds trivially. A (sub)term with function symbol f_g is necessarily a descendant of a term introduced by rule $m1$. Such a term is a redex for $m4$, and potentially also for $m2$. Five things can happen to the descendants: they may be contracted according to $m2$ or $m4$, they may persist in t, they may be duplicated and they may be deleted by contraction of a higher redex. Duplication and deletion are trivially copied in R. If the redex persists in t, we can construct t' by applying $m4$, which brings us to the case where we have a (sub)term with function symbol f^S (see below). The result of a contraction according to $m2$ can be obtained in R by contracting the original term according to $m3$. The result of a contraction according to $m4$ brings us again to the case where we have a (sub)term with function symbol f^S. Apart from the two cases above, a (sub)term with function symbol f^S can be introduced by $s1$. Because of simple completeness, such a (sub)term is a redex of at least one of the rules in *Other*, but there can only be root-overlap with other rules, because f^S does not occur in Σ. Therefore, the descendants of this redex may be duplicated, deleted, or contracted according to one of the rules in *Other*. Duplication and deletion are again trivially copied in R, contraction according to $o1$ or $o3$ corresponds to a contraction according to $o2$ or $o3$, respectively. Finally, if a (sub)term with f^S is left in t, we have $\neg\mathcal{D}_S(t)$, but we have tR'^*t' by a most general rule in *Other*. By the argument above, we now have that sRt'

Termination Conservation Obvious by induction on $NVP(R)$.

■

3.7.2 Transforming Complicated rhss

Here we present a transformation that will transform a TRS N, which may have most general rules with rhss that do not conform to the restrictions imposed on MTRSs, into a simulating MTRS M.

Any most general rule with a rhs that does not conform to the restrictions imposed on MTRSs can be brought in one of three forms:

1. $l(\vec{x}) \to x_i$. A rule with a single variable on the rhs.
2. $l(\vec{x}) \to c(\vec{t})$, where c is a constructor.
3. $l(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, u, \vec{t}, \vec{z})$, where h is a defined function. A rule which replaces the variables $\vec{y}$ by terms $u, \vec{t}$, where we take $\vec{x}$ and $\vec{z}$ of maximal length, and where u is either a variable (not equal to the last variable of $\vec{y}$) or a term $g(\vec{u})$.

The first case is simulated by the following simulation:

$$l(\vec{x}, y, \vec{z}) \to \mathbf{return}(y) \tag{3.23}$$

$$\mathbf{return}(y) \to y \tag{3.24}$$

where $|x| = i - 1$, rule (3.24) is a rule of form **R**, and rule (3.23) falls in the class that is dealt with below. The symbol **return** and its defining rule (3.24) can be reused for other occurences of this case.

Rule 3.24 is also used for the simulation of the second case with the rule

$$l(\vec{x}) \to \mathbf{return}(c(\vec{t})) \tag{3.25}$$

In the third case, the goal is to reduce the non-compliant segment $u, \vec{t}$.

In case u is a variable, we replace the rule by the following rules:

$$l(\vec{x}, \vec{y}, \vec{z}) \to h^R(\vec{x}, u, \vec{y}, \vec{z}) \tag{3.26}$$

$$h^R(\vec{x}, u', \vec{y}, \vec{z}) \to h(\vec{x}, u', \vec{t}, \vec{z}), \tag{3.27}$$

Where u' is a fresh variable. Rule (3.26) is an instance of **A**, and rule (3.27) has a shorter non-compliant segment $\vec{t}$.

In case $u = g(\vec{u})$, we replace the rule by the following rules:

$$l(\vec{x}, \vec{y}, \vec{z}) \quad \to \quad h^R(\vec{x}, \vec{u}, \vec{t}, \vec{z}) \tag{3.28}$$

$$h^R(\vec{x}, \vec{y'}, \vec{z'}) \quad \to \quad h(\vec{x}, g(\vec{y'}), \vec{z'}) \tag{3.29}$$

where $\mid \vec{z'} \mid = \mid \vec{t} \mid + \mid \vec{z} \mid, \mid \vec{y'} \mid = \mid \vec{u} \mid$; h^R is a fresh function symbol which did not already occur in the TRS; and $\vec{y'}$ and $\vec{z'}$ consist entirely of fresh variables. Rule (3.28) contains one function symbol less than the original rule, and rule (3.29) is an instance of **C**.

We take the simulation map $\mathcal{S}$ to be undefined for h^R and **return**. Repeated application of the transformations above to a TRS with minimal lhs's leads to an MTRS.

We note that the four transformations above fulfill the requirements for a simple simulation (see Lemma 2), so the rhs transformation yields a simulating TRS.

3.7.3 Simulating General Left-Linear TRSs by Simply Complete left-linear TRSs

Until now, we have only dealt with simply complete TRSs. Unfortunately, simple completeness is a rare property. Here we will show that, under the restriction to innermost rewriting with syntactic specificity ordering, every TRS can be simulated by a simply complete TRS.

Let the TRS $\langle \Sigma, R \rangle$ be given, and let $\Sigma_p \subseteq \Sigma$ be the set of function symbols for which R has no most general rule. Let Σ_c contain a so-called *constructor variant* f_c for every $f \in \Sigma_u$, and let $\mathcal{S}(f_c) = f$. Given a term t or a sequence $\vec{t}$, define t_c or $\vec{t}_c$ to be the term or sequence obtained by replacing all $f \in \Sigma_u$ by their constructor variants f_c. Taking $R' = \{o1 : f(\vec{t}_c) \to s | f(\vec{t}) \in R\} \cup \{c1 : f(\vec{x}) \to f_c(\vec{x}) | f \in \Sigma_u\}$, we have obtained a simply complete TRS $\langle \Sigma \cup \Sigma_c, R' \rangle$.

Proposition 4 *The simulation map $\mathrm{I}_{\mathcal{F}}$ induces a simulation of R by R'.*

Proof We have to prove soundness, completeness and termination conservation.

Soundness We observe that, given a rewrite sequence $t_1 \rightarrow t_2 \rightarrow \ldots t_n$ in R', it follows that either $\mathcal{S}(t_i) = \mathcal{S}(t_{i+1})$ (in case $c1$ is applied), or $\mathcal{S}(t_i) R \mathcal{S}(t_{i+1})$ (in case $o1$ is applied), so always $t_1 R * t_n$.

Completeness We observe that a step $\mathcal{S}(t_i) R \mathcal{S}(t_{i+1})$ may not be possible in R' because some subterms of t_i have an original function symbol where a constructor variant is needed. We may first rewrite exactly these subterms with rules of type $c1$, however, $t_i R' * t'_i$, with $\mathcal{S}(t'_i) = \mathcal{S}(t_i)$, and then we have $t'_i R' t_{i+1}$. Because a step $t_i R t_{i+1}$ is only taken when all subterms of t_i are already in normal form, rewriting t_i to t'_i does not invalidate future R-rewrites and because of specificity ordering, rules of type $c1$ can only rewrite terms t for which $\mathcal{S}(t)$ is a normal form.

Termination Conservation Conservation of termination follows from the fact that only a finite number of applications of rules of type $c1$ is possible on any term, so if there is an infinite reduction on t according to R', there is necessarily an infinite reduction on $\mathcal{S}(t)$ according to R.

■

3.7.4 Efficiency Considerations

The efficiency of compilers can be expressed by several measures:

- The size (in number of rules) of the target program;
- The time and space taken for compilation from source to target language;
- The time and space taken by an execution of the target program, compared to the time and space taken by execution of the source program.

For our translation, it is clear that the size of the target program depends in a linear fashion on the total number of occurrences of function symbols in the source program, and rules in the target program are at least as simple as the rules in the source program.

With regard to the space taken by the compilation, we observe that the number of new rules constructed depends in a linear fashion on the total number of occurrences of function symbols in the source program. Thus, a naive implementation needs at most an amount of space linear in the size of the source program.

With respect to the time taken, even a naive implementation of **sim** that scans all rules to find a rule with nonvariable arguments, will only be quadratic in the number of rules and linear in the number of symbol occurrences in lhs's.

Considering the execution time of the target program, we see that the number of rewriting steps is linearly increased by the compilation. The complexity of executing a single step, however, is decreased. In practice, this leads to comparable performance.

3.8 Related Work

In [FM91], Fradet and Le Métayer advocate that the implementation process of a functional language should be described in a purely functional way. Their compilation technique is presented as a closure-based continuation-passing transformation on lambda terms. They argue that staying entirely within the lambda calculus yields a compiler that is easier to understand and prove correct, just as we argue that staying entirely within (first-order) rewriting systems makes life easier. Our use of first-order TRSs instead of the lambda calculus allows us to be less explicit about closures and continuations (closures are not needed at all, and continuations are made explicit only when MTRSs are interpreted as ARM programs) and to give a natural implementation of pattern matching.

In [HG94], a provably correct compiler for term rewriting systems is described, using an abstract machine TRIM, which appears to be about as simple ARM. The approach seems to be geared more towards provability than towards efficiency, because environments are built explicitly on the heap (whereas the 'environments' of ARM are on a, cheaper, stack).

In the context of (lazy) functional languages, many different abstract machines are used, notably SKIM [Tur79], the Categorial Abstract Machine (CAM, [CCM85]), the Three Instruction Machine (TIM, [FW87]), the G-machine [PJ87], its successor, the spineless tagless G-machine (STG, [JS89]), and the ABC machine [PvE93]. These machines address lazy graph rewriting of curried higher-order function applications (CAM is basically innermost, but supports lazy evaluation). In contrast, ARM is designed for first-order innermost reduction on stacks, where the graph structure is only explicit in the normal forms, pointers to which reside on the C and A stacks (see Fig. 3.6). Laziness can be added as a source-to-source transformation, given one extra ARM instruction (see Chapter 6). Higher-order functions as they appear in implementations of functional programming languages can be implemented by *applicative term rewriting systems* [Tur79].

Because lazy graph rewriting is expensive, and most of the time not needed, most of the lazy functional abstract machines have add-ons for innermost (strict) rewriting, making them more complicated than ARM.

It is our experience that comparisons of (implementations of) abstract machines tend to be hard to interpret and are often misleading. A somewhat specific comparison is presented in [HF$^+$96]. Based on that and other experiences ARM technology can be said to lead to efficient implementations.

The abstract machine presented in Section 3.4 is much less complex than the version of ARM presented in [KW93b], and somewhat less complex than the improved version of ARM described in [WK94]. The most important difference with the latter lies in the fact that it is not possible anymore to push variables or terms onto the control stack.

3.9 Conclusions and Future Work

We have presented minimal term rewriting systems (MTRSs), and shown that an MTRS can directly be interpreted as a program for the Abstract Rewriting Machine (ARM), which has a straightforward, efficient implementation on conventional hardware.

We have also presented a compiler from left-linear TRSs into simulating MTRSs. The transformations can be expressed in a concise way, and their correctness proofs are short and easy to grasp. Furthermore, the transformations are described as executable specifications, which can be used as an efficient compiler. The resulting code is similar to the code generated by an earlier version of our TRS compiler, with which favorable results have been reached [HF$^+$96].

In chapter 6, we present a transformation to simulate lazy rewriting by eager (innermost) rewriting. It appears that this transformation can be simplified greatly by first applying the transformations in this chapter, and then the laziness transformation, which is much simpler when only MTRSs have to be considered.

Similarly, we expect that the transformations given in this chapter could simplify other research on TRSs (e.g., Hans Zantema suggested that termination proofs might be simpler after our transformations, but we have not yet investigated this issue in any depth).

In the future, we hope to find a bigger class of TRSs for which a strategy-independent simulation by MTRSs can be given. For our present implementation requirements, however, the current class is sufficient.

An interesting class of TRSs (which unfortunately has no inclusion relation with simply complete TRSs) is the class that admits *specificity ordering* as defined in [BBKW89]. It appears that applying the transformation in this chapter to a member of this class yields a simulating MTRS, if we consider the priority rewrite relation, without any further assumption about the strategy. We would like to establish this rigorously.

Chapter 4

Formal Specification of EPIC, a Language for Term Rewriting

In Chapter 3, we have defined an abstract rewriting machine ARM, Minimal Term Rewriting Systems (MTRSs), which can be viewed as ARM programs, and a compilation algorithm from Term Rewriting Systems (TRSs) into MTRSs. In order to actually use all this, we need to define a language to express TRSs in. In this chapter, we present Epic, an intermediate language for the expression of term rewriting systems. We provide abstract syntax, static and operational semantics, and one possible concrete syntax of Epic.

4.1 Introduction

The Abstract Reduction Machine (ARM) is a simple, efficient platform for the execution of term (graph) rewriting systems. In Chapter 3, we have shown how general TRSs can be turned into ARM programs, under the assumption of innermost rewriting with syntactic specificity. In order to use this theoretical exercise for practical software, we need an intermediate language to express TRSs.

In this chapter, we present such a language for the expression of TRSs, called Epic. In order to keep the implementation effort and the size of the resulting software and documentation manageable, we have attempted to keep the built-in functionality minimal. Rather than extending Epic itself, we envision extended languages implemented by translation into Epic.

4.1.1 EPIC in a nutshell

Epic features innermost rewriting with syntactic specificity ordering [BBKW89]; a more specific rule has higher precedence than a more general rule. It supports external datatypes and separate compilation of modules.

An Epic module consists of a signature and a set of rules. The signature declares functions, each with an arity (number of arguments). In addition, functions can be declared *external* (i.e., defined in another module, or directly in C), or *free* (i.e., not defined in any module). The rules are left-linear rewrite rules.

The choice for innermost rewriting is a consequence of efficiency considerations; the average time per reduction for an innermost implementation is less than that for other strategies. Alternative strategies can be implemented by transformations of TRSs. E.g., in Chapter 6 a method is described which makes lazy rewriting available on an innermost implementation.

The choice for syntactic specificity ordering is a compromise between semantic elegance (no ordering between rules) and practical predictability (a total ordering of rules). By virtue of being *partial*, syntactic specificity ordering makes EPIC a nondeterministic language. In contrast to languages with *don't know nondeterminism* (i.e., the implementation is required to explore all choices) such as Prolog, EPIC is a language with *don't care nondeterminism* (i.e., the programs should be constructed in such a way, that the choice does not matter). Unfortunately, this means that for a non-confluent TRS, the result of a computation depends on an arbitrary decision by the compiler. An EPIC compiler should emit a warning when this occurs.

EPIC is intentionally defined on the level of abstract syntax: when using EPIC as a target language, generating abstract syntax directly (as an internal data structure, or in a simple textual format) avoids producing and parsing concrete syntax. In addition, the abstract syntax is representation independent. Thus, provided programs in some language L can be interpreted as TRSs, the abstract syntax of L can be used without translation as the abstract syntax of EPIC by providing an interface.

EPIC's type-system is liberal: it is single-sorted, requiring only the usual restrictions for TRSs (the LHS of a rule is not a sole variable; all arities coincide; and a variable must be instantiated – in the LHS – before it is used), and some concerning modules (free and external functions may not become defined). EPIC's tool set contains a type-checker which verifies these requirements.

4.1.2 The EPIC tool set

In order to be maximally portable, the EPIC tool set is implemented in EPIC itself. The EPIC tool set includes the following tools:

- An EPIC parser;
- An EPIC type-checker;
- A printer for parsed EPIC programs;
- A printer for ARM code;
- A source-to-source transformer adding syntactic sugar for a let-like construct to EPIC.
- A compiler which translates EPIC to ARM.

- The ARM interpreter.
- A compiler from ARM to C functions, one for each function in the original TRS. These functions can be linked, statically, to the interpreter.

The EPIC tool set is available via WWW at `http://www.cwi.nl/epic/`.

4.1.3 Example EPIC programs

As mentioned, the concrete syntax of EPIC is not very relevant. In Section 4.4.2, we will define one concrete syntax (which is the one we use), but we do not propose that syntax to be 'the' concrete syntax of EPIC; it has none. To provide a first taste of EPIC, however, concrete syntax must be used.

We show a module defining binary numbers with addition (`plus`) and multiplication (`times`). This module is an instance of the scheme published as [WZ95]. The free constructors `o` and `i` represent binary zero and one, respectively, and `jxt` (mnemonic for *juxtaposition*) is used to concatenate binary numbers. Note that `jxt` is not free; there is a rule defining the removal of initial zeroes, and a rule expressing concatenation of composite numbers in terms of addition and concatenation.

```
module Numbers
types
  o: -> Nat {free}; i: -> Nat {free}; jxt: Nat # Nat -> Nat;
  plus: Nat # Nat  -> Nat; times: Nat # Nat -> Nat;
rules
  jxt(o,X) = X;
  jxt(X,jxt(Y,Z)) = jxt(plus(X,Y),Z);
  plus(o,X) = X;   plus(i,o) = i ;
  plus(i,i) = jxt(i,o);
  plus(i,jxt(X,Y)) = jxt(X,plus(i,Y));
  plus(jxt(X,Y),Z) = jxt(X,plus(Y,Z));
  times(o,X) = o ;   times(i,X) = X;
  times(jxt(X,Y),Z) = jxt(times(X,Z),times(Y,Z));
```

In an other module, the functions defined in this module can be used by declaring them with the property *external*:

```
module UseNumbers
types
  o: -> Nat {external}; i: -> Nat {external}; jxt: Nat # Nat -> Nat {external};
  plus: Nat # Nat -> Nat {external}; times: Nat # Nat  -> Nat {external};
  square: Nat -> Nat; cube: Nat -> Nat;
rules
  square(X) = times(X,X);
  cube(X) = times(X,square(X,X));
```

The modules above can be compiled separately, after which the code can be combined to yield a tool that normalizes arithmetic expressions into their values.

4.2 Abstract Syntax

The *abstract* syntax of Epic defines essential structural information, void of representational aspects. We define the abstract syntax as a collection of sorts (corresponding to classes of objects) and functions (the information that can be retrieved from objects), and a number of properties (relationships between objects and functions). This leaves the abstract syntax underspecified: the signature is only partly given, and interpretations that identify certain functions on certain domains are allowed.

We do not want to restrict to initial models, because this would hamper future extensions of Epic, but the current specification very likely admits many unacceptable models among the intended ones. More research is needed to find the right class of models.

Compared to the traditional approach (i.e., defining a graph/tree language as abstract syntax), our approach has the advantage that the abstract syntax remains truly abstract; essential aspects are defined, and irrelevant detail is avoided. That is, the concrete representation is completely hidden. Thus, the abstract syntax representation of existing languages may be reused by providing an interface. A disadvantage is that for any concrete use of Epic, one should be explicit about the concrete interpretation of the abstract syntax. In Section 4.4 we present one particular term model of Epic's abstract syntax, which we use in our Epic compiler.

In this document we indicate specification segments with bars to their left: a single bar signifies syntax (sorts and functions); a double bar signifies semantical information.

First, we define the sorts. The sort `Id` contains the names of functions, modules and variables. It is a parameter of Epic, we only require that sufficiently many names are available, and that comparison between names is possible.

We use `Number` to designate the arity of functions. `Number` need not be the set of natural numbers $\mathbb{N}$ (which is infinite), although, in practice, sufficiently many distinct numbers should exist.

The status of the other sorts does not need further explanation:

`Prog` — An Epic program
`Mod` — An Epic module
`Type` — The type of a function
`Rule` — A rewrite rule
`Term` — A term
$\mathtt{Sq_m}$,$\mathtt{Sq_f}$,$\mathtt{Sq_r}$,$\mathtt{Sq_t}$, — Sequences
`Id` — Identifiers
`Number` — Numbers

In the remainder of this chapter, all formulae are implicitly universally quantified, where the name of variables (possibly with subscript) indicates their range: p for `Prog`, m for `Mod`, f for `Type`(f for function-type), r for `Rule`, t for `Term`, n for `Id`, α for `Number` and i for `Sq` (and, for example, i^t for $\mathtt{Sq_t}$).

We introduce various auxiliary sorts and (overloaded) functions in order to reduce the total number of functions and equations, or to reduce trivial conditions. The meaning of a formula is the set of instances that are well-typed using base (i.e., non auxiliary) sorts. We do not consider sub-sorts.

Predicates are boolean valued, total functions. Their use in a condition or consequence signifies truth; their negation (e.g., $\neg\mathtt{is\_var}(\mathtt{lhs}(r))$, or $t \notin r$) signifies falsehood. We assume and use some degree of initiality for predicates: if the value of a predicate isn't defined to be true, then it is taken to be false.

We use the notation $\langle\ldots\rangle$ for tuples (i.e., members of cartesian products). For example, if a and b are of sort A and B, respectively, then $\langle a,b\rangle$ is of sort $A\#B$.

Finally, we take recursively enumerable sets to be a primitive.

Let $\mathtt{Sq} = \mathtt{Sq}_\mathtt{m} \cup \mathtt{Sq}_\mathtt{s} \cup \mathtt{Sq}_\mathtt{r} \cup \mathtt{Sq}_\mathtt{t}$ be the sort of all sequences.

`subs`$_\mathtt{m}$:	`Prog`	`->` $\mathtt{Sq}_\mathtt{m}$	The sequence of modules in the program
`at`:	$\mathtt{Sq}_\mathtt{m}$	`-> Mod`	The first module of a sequence
`adv`:	$\mathtt{Sq}_\mathtt{m}$	`->` $\mathtt{Sq}_\mathtt{m}$	The rest of the modules of a sequence
`subs`$_\mathtt{f}$:	`Mod`	`->` $\mathtt{Sq}_\mathtt{f}$	The sequence of a functions in the module
`at`:	$\mathtt{Sq}_\mathtt{f}$	`-> Type`	The first function of a sequence
`adv`:	$\mathtt{Sq}_\mathtt{f}$	`->` $\mathtt{Sq}_\mathtt{f}$	The rest of the functions of a sequence
`subs`$_\mathtt{r}$:	`Mod`	`->` $\mathtt{Sq}_\mathtt{r}$	The sequence of rules in the module
`at`:	$\mathtt{Sq}_\mathtt{r}$	`-> Rule`	The first rule of a sequence
`adv`:	$\mathtt{Sq}_\mathtt{r}$	`->` $\mathtt{Sq}_\mathtt{r}$	The rest of the rules of a sequence

`name`:	`Type`	`-> Id`	the name of a function
`arity`:	`Type`	`-> Number`	The number of arguments a function takes
`external`:	`Type`	*predicate*	Is the function external
`free`:	`Type`	*predicate*	Is the function (globally) free

`lhs`:	`Rule`	`-> Term`	The *lhs* of the rule
`rhs`:	`Rule`	`-> Term`	The *rhs*

`ofs`:	`Term`	`-> Id`	The outermost function symbol
`subs`$_\mathtt{t}$:	`Term`	`->` $\mathtt{Sq}_\mathtt{t}$	The sub-terms of the term
`at`:	$\mathtt{Sq}_\mathtt{t}$	`-> Term`	The first term in a sequence of terms
`adv`:	$\mathtt{Sq}_\mathtt{t}$	`->` $\mathtt{Sq}_\mathtt{t}$	The rest of the terms
`is_var`:	`Term`	*predicate*	Is the term a variable

`null`:	`Sq`	*predicate*	is this sequence empty

0:		`-> Number`	The number zero
_+1:	`Number`	`-> Number`	Successor function

Domains

We do not require all functions to be total; the functions `ofs` and `subs`$_\mathtt{t}$ are not defined on variables, and the functions `at` and `adv` are only defined on non-null sequences.

$$\mathtt{is\_var}(t) \Longrightarrow t \notin \mathit{dom}(\mathtt{ofs}) \wedge t \notin \mathit{dom}(\mathtt{subs}_\mathtt{t})$$
$$\neg\mathtt{null}(i) \Longrightarrow i \in \mathit{dom}(\mathtt{adv}) \wedge i \in \mathit{dom}(\mathtt{at})$$

4.3 Semantics

In order to define static and operational semantics, some auxiliary notions are needed, which we will first introduce.

Arity

The arity is the number of elements in a sequence.

$\mathtt{arity}\colon\ \mathtt{Sq}\ \ \mathtt{->}\ \mathtt{Number}$

$\mathtt{null}(i) \Longrightarrow \mathtt{arity}(i) = 0$

$\neg\mathtt{null}(i) \Longrightarrow \mathtt{arity}(i) = \mathtt{arity}(\mathtt{adv}(i)) + 1$

Containment

Let $\mathtt{Mod}^I = \mathtt{Mod} \cup \mathtt{Sq_m}$, $\mathtt{Type}^I = \mathtt{Type} \cup \mathtt{Sq_f}$, $\mathtt{Rule}^I = \mathtt{Rule} \cup \mathtt{Sq_r}$ and $\mathtt{Term}^I = \mathtt{Term} \cup \mathtt{Sq_t}$ be the union of structures and their sequences; let $\mathtt{Struct} = \mathtt{Prog} \cup \mathtt{Mod} \cup \mathtt{Type} \cup \mathtt{Rule} \cup \mathtt{Term}$ be the set of all structures; and let $\mathtt{Struct}^I = \mathtt{Prog} \cup \mathtt{Mod}^I \cup \mathtt{Type}^I \cup \mathtt{Rule}^I \cup \mathtt{Term}^I$.

$\epsilon\colon\ \mathtt{Struct}^I\ \#\ \mathtt{Struct}^I$ *predicate*

$x_1 \epsilon x_2 \wedge x_2 \epsilon x_3 \Longrightarrow x_1 \epsilon x_3$

$x \epsilon x$

$\mathtt{lhs}(r) \epsilon r$

$\mathtt{rhs}(r) \epsilon r$

$\mathtt{subs}(t) \epsilon t$

$\neg\mathtt{null}(i) \Longrightarrow \mathtt{adv}(i) \epsilon i \wedge \mathtt{at}(i) \epsilon i$

Substitutions

Let $\mathtt{Var} = \{t | \mathtt{is\_var}(t)\}$ be the set of all variables, let v, possibly with sub-script, range over $\mathtt{Var}$, let $\mathtt{Subst} = \mathcal{P}(\mathtt{Var}\#\mathtt{Term})$ be the set of variable-value pairs which homomorphically generate substitutions, and let σ, possibly with subscript, range over $\mathtt{Subst}$.

$\_^{\_}\colon\ \mathtt{Term}^I\ \#\ \mathtt{Subst}\ \ \mathtt{->}\ \mathtt{Term}^I$ (e.g. t^σ)

$\langle v, t\rangle \in \sigma \Longrightarrow v^\sigma = t$

$\neg\mathtt{is\_var}(t) \Longrightarrow \mathtt{ofs}(t^\sigma) = \mathtt{ofs}(t)$

$\mathtt{subs_t}(t^\sigma) = \mathtt{subs_t}(t)^\sigma$

$\mathtt{null}(i) \Longrightarrow \mathtt{null}(i^\sigma)$

$\neg\mathtt{null}(i) \Longrightarrow \mathtt{at}(i^\sigma) = \mathtt{at}(i)^\sigma$

$\neg\mathtt{null}(i) \Longrightarrow \mathtt{adv}(i^\sigma) = \mathtt{adv}(i)^\sigma$

Contexts

Containment can not be used to express the *position* of sub-terms.

We use the slightly operational notion of *contexts* [Klo92] to express position. With contexts, one can use containment to reason about positions.

Intuitively, a context is a structure with a hole in it. We define contexts by extending the set of terms with the hole (□). Unlike [Klo92], we take □ to be a variable; this allows us to use substitution for context instantiation.

$$\begin{array}{|ll} \Box\text{:} & \texttt{-> Term} \end{array}$$

$$\begin{array}{||l} \quad \texttt{is\_var}(\Box) \end{array}$$

Let `Context` be the set of rules and terms, and their sequences, which contain exactly one occurrence of □. We forego the constructive definition of `Context`, which is trivial but tedious. Let $\texttt{Context}\cap\texttt{Term} = \texttt{Context}^T$, $\texttt{Context}\cap\texttt{Rule} = \texttt{Context}^R$, and $\texttt{Context}\cap\texttt{Sq} = \texttt{Context}^S$, and let γ, γ^t, γ^r and γ^i range over `Context`, $\texttt{Context}^T$, $\texttt{Context}^R$ and $\texttt{Context}^S$, respectively.

Instantiation of a context coincides with substitution of the hole.

$$\begin{array}{|llll} \_[\_]\text{:} & \texttt{Context}^R & \texttt{\# Term} & \texttt{->Rule} \\ & \texttt{Context}^T & \texttt{\# Term} & \texttt{->Term} \\ & \texttt{Context}^S & \texttt{\# Term} & \texttt{->Sq} \\ & \texttt{Context} & \texttt{\# Context} & \texttt{->Context} \end{array}$$

$$\begin{array}{||l} \quad \texttt{lhs}(\gamma^r[t]) = \texttt{lhs}(\gamma^r)[t] \\ \quad \texttt{rhs}(\gamma^r[t]) = \texttt{rhs}(\gamma^r)[t] \\ \quad \gamma^t[t] = \gamma^{t\{\langle\Box,t\rangle\}} \end{array}$$

Two contexts are compatible if they can be instantiated to the same

$$\begin{array}{|lll} \sim\text{:} & \texttt{Context \# Context} & \textit{predicate} \end{array}$$

$$\begin{array}{||l} \quad \gamma_1[t_1] = \gamma_2[t_2] \Longrightarrow \gamma_1 \sim \gamma_2 \end{array}$$

In order to define rightmost innermost reduction in Section 4.3.2, we define a pre-order on contexts: if two contexts are compatible, and □ occurs above or 'to the left' (picturing **adv** as movement to the right), then that context is smaller in pre-order.

$$\begin{array}{|lll} <\text{:} & \texttt{Context \# Context} & \textit{predicate} \end{array}$$

$$\begin{array}{||l} \quad \gamma_1 < \gamma_2 \wedge \gamma_2 < \gamma_3 \Longrightarrow \gamma_1 < \gamma_3 \\ \quad \gamma_1[\gamma^t] = \gamma_2 \Longrightarrow \gamma_1 < \gamma_2 \\ \quad {\gamma^r}_1 \sim {\gamma^r}_2 \wedge \Box \in \texttt{lhs}({\gamma^r}_1) \wedge \Box \in \texttt{rhs}({\gamma^r}_2) \Longrightarrow {\gamma^r}_1 < {\gamma^r}_2 \\ \quad {\gamma^i}_1 \sim {\gamma^i}_2 \wedge \neg\texttt{null}({\gamma^i}_2) \wedge \Box \in \texttt{at}({\gamma^i}_1) \wedge \Box \in \texttt{adv}({\gamma^i}_2) \Longrightarrow {\gamma^i}_1 < {\gamma^i}_2 \end{array}$$

Matching

The predicate `matches` is used to determine whether there is a match between two terms, and if there is a match, `match` delivers the matching substitution.

$$\begin{array}{|llll} \texttt{matches:} & \texttt{Term}^I \# \texttt{Term}^I & \textit{predicate} \\ \texttt{match:} & \texttt{Term}^I \# \texttt{Term}^I & \texttt{-> Subst} \end{array}$$

$\mathtt{matches}(s, v)$
$\neg\mathtt{is\_var}(t_1) \wedge \neg\mathtt{is\_var}(t_2) \wedge \mathtt{ofs}(t_1) = \mathtt{ofs}(t_2) \wedge \mathtt{matches}(\mathtt{subs_t}(t_1), \mathtt{subs_t}(t_2))$
$\quad \Longrightarrow \mathtt{matches}(t_1, t_2)$
$\mathtt{null}(i_1) \wedge \mathtt{null}(i_2) \vee (\mathtt{matches}(\mathtt{at}(i_1), \mathtt{at}(i_2)) \wedge \mathtt{matches}(\mathtt{adv}(i_1), \mathtt{adv}(i_2)))$
$\quad \Longrightarrow \mathtt{matches}(i_1, i_2)$
$\mathtt{match}(s, v) = \{\langle v, s\rangle\}$
$\neg\mathtt{is\_var}(t_1) \wedge \neg\mathtt{is\_var}(t_2) \wedge \mathtt{ofs}(t_1) = \mathtt{ofs}(t_2) \wedge \mathtt{matches}(\mathtt{subs_t}(t_1), \mathtt{subs_t}(t_2))$
$\quad \Longrightarrow \mathtt{match}(t_1, t_2) = \mathtt{match}(\mathtt{subs_t}(t_1), \mathtt{subs_t}(t_2))$
$\mathtt{null}(i_1) \wedge \mathtt{null}(i_2)) \Longrightarrow \mathtt{match}(i_1, i_2) = \{\}$
$\neg\mathtt{null}(i_1) \wedge \neg\mathtt{null}(i_2) \wedge \mathtt{matches}(\mathtt{at}(i_1), \mathtt{at}(i_2)) \wedge \mathtt{matches}(\mathtt{adv}(i_1), \mathtt{adv}(i_2))$
$\quad \Longrightarrow \mathtt{match}(i_1, i_2) = \mathtt{match}(\mathtt{at}(i_1), \mathtt{at}(i_2)) \cup \mathtt{match}(\mathtt{adv}(i_1), \mathtt{adv}(i_2))$

Syntactic Specificity ordering

Intuitively, any non-variable term is more specific than a variable. This is the basis for a partial order on terms: syntactic specificity. The order is extended on rules.

$\prec$: `Rule # Rule` *predicate*
`Term # Term` *predicate*
`Sq # Sq` *predicate*
$\preceq$: `Term # Term` *predicate*
`Sq # Sq` *predicate*

$\mathtt{lhs}(r_1) \prec \mathtt{lhs}(r_2) \Longrightarrow r_1 \prec r_2$
$\neg\mathtt{is\_var}(t) \Longrightarrow v \prec t$
$\neg\mathtt{is\_var}(t_1) \wedge \neg\mathtt{is\_var}(t_2) \wedge \mathtt{ofs}(t_1) = \mathtt{ofs}(t_2) \wedge \mathtt{subs_t}(t_1) \prec \mathtt{subs_t}(t_2) \Longrightarrow t_1 \prec t_2$
$\mathtt{null}(i_1) \wedge \mathtt{null}(i_2) \Longrightarrow i_1 \preceq i_2$
$\neg\mathtt{null}(i_1) \wedge \neg\mathtt{null}(i_2) \wedge \mathtt{at}(i_1) \prec \mathtt{at}(i_2) \wedge \mathtt{adv}(i_1) \preceq \mathtt{adv}(i_2) \Longrightarrow i_1 \prec i_2$
$\neg\mathtt{null}(i_1) \wedge \neg\mathtt{null}(i_2) \wedge \mathtt{at}(i_1) \preceq \mathtt{at}(i_2) \wedge \mathtt{adv}(i_1) \prec \mathtt{adv}(i_2) \Longrightarrow i_1 \prec i_2$
$x_1 \prec x_2 \Longrightarrow x_1 \preceq x_2$
$x \preceq x$
$v_1 \preceq v_2$

4.3.1 Static Semantics

EPIC programs should satisfy the following semantic requirements:

(i) A function should be defined in one module only (it can be used in more than one module). *This restriction is a consequence of implementational aspects, and should be removed in later versions of* EPIC;

(ii) A function that is declared to be *free* should never become defined;

(iii) A function that is declared to be *external* in a module should not become defined in that module;

(iv) The number of immediate sub-terms of a term must be in accordance with the arity of the outermost function symbol of that term;

(v) The left-hand side of a rewrite rule should not be a sole variable;

(vi) All variables occurring in the right-hand side should also occur in the left-hand side.

(vii) Rules must be left-linear.

This is expressed by the following formulae:

$$m_1 \in p \land m_2 \in p \land r_1 \in m_1 \land r_2 \in m_2 \land \mathtt{ofs}(\mathtt{lhs}(r_1)) = \mathtt{ofs}(\mathtt{lhs}(r_2)) \Longrightarrow m_1 = m_2 \tag{i}$$

$$r \in p \land f \in p \land \mathtt{ofs}(\mathtt{lhs}(r)) = \mathtt{name}(f) \Longrightarrow \neg\mathtt{free}(f) \tag{ii}$$

$$r \in m \land s \in m \land \mathtt{ofs}(\mathtt{lhs}(r)) = \mathtt{name}(f) \Longrightarrow \neg\mathtt{external}(f) \tag{iii}$$

$$t \in p \land f \in p \land \mathtt{ofs}(t) = \mathtt{name}(f) \Longrightarrow \mathtt{arity}(f) = \mathtt{arity}(\mathtt{subs}_{\mathtt{t}}(t)) \tag{iv}$$

$$\neg\mathtt{is\_var}(\mathtt{lhs}(r)) \tag{v}$$

$$v \in \mathtt{rhs}(r) \Longrightarrow v \in \mathtt{lhs}(r) \tag{vi}$$

$$\gamma[v] = \mathtt{lhs}(r) \Longrightarrow v \notin \gamma \tag{vii}$$

4.3.2 Operational Semantics

An EPIC implementation is a procedure which, given a term and a program, attempts to determine a normal form of that term that can be reached with rightmost innermost reduction and in accordance with syntactic specificity (i.e., given a rightmost innermost redex, a most-specific rule must be applied to it).

Rightmost innermost reduction and syntactic specificity ordering do not make a rewrite system deterministic: unordered rules, or rules of equal specificity can be applicable to the same redex. Accordingly, we must consider *sets* of reducts and normal forms.

$$\begin{array}{lll}
\mathtt{potentials:} & \mathtt{Term\ \#\ Prog} & \mathtt{->}\ \mathcal{P}(\mathtt{Context}\ \#\ \mathtt{Term}\ \#\ \mathtt{Rule}) \\
\mathtt{reducts:} & \mathtt{Term\ \#\ Prog} & \mathtt{->}\ \mathcal{P}(\mathtt{Term}) \\
\mathtt{normal\_forms:} & \mathtt{Term\ \#\ Prog} & \mathtt{->}\ \mathcal{P}(\mathtt{Term})
\end{array}$$

$$\begin{array}{l}
\mathtt{potentials}(t_1, p) = \{\langle\gamma, t_2, r\rangle \mid r \in p \land \gamma[t_2] = t_1 \land \mathtt{matches}(t_2, \mathtt{lhs}(r))\} \\
\mathtt{reducts}(t_1, p) = \\
\qquad \{\gamma[\mathtt{rhs}(r)^{\mathtt{match}(t_2,\mathtt{lhs}(r))}] \mid \exists\langle\gamma, t_2, r\rangle \in \mathtt{potentials}(t_1, p) : \\
\qquad\qquad\qquad \neg\exists\langle\gamma', t', r'\rangle \in \mathtt{potentials}(t_1, p) : \gamma < \gamma' \lor r \prec r'\} \\
\mathtt{reducts}(t, p) = \emptyset \Longrightarrow \mathtt{normal\_forms}(t, p) = \{t\} \\
\mathtt{reducts}(t_1, p) \neq \emptyset \Longrightarrow \mathtt{normal\_forms}(t_1, p) = \bigcup_{t_2 \in \mathtt{reducts}(t_1,p)} \mathtt{normal\_forms}(t_2, p)
\end{array}$$

An implementation is a procedure which, given a program p and a term t_0, may or may not terminate. If it terminates, it yields a member t_n of $\mathtt{normal\_forms}(t_0, p)$.

The choice for innermost reduction is inspired by efficiency considerations. The fact that in innermost reduction, the arguments of a redex are already in normal form can be used in three different ways:

1. Recomputations of terms reappearing on the RHS can be prevented.

2. Only the subterms that are known to be in normal form need to be represented in (expensive) heap space, the parts of the term that should still be examined can be represented on (cheap) stack space.

3. The representation on stacks simplifies the procedure to find the next redex.

This choice is not really felt as a limitation when EPIC is used as a target language; based on an innermost implementation with syntactic specificity ordering, alternative strategies can be simulated (for an example, see Chapter 6).

The choice for syntactic specificity ordering is a compromise between semantic elegance and predictability of practical behaviour. Totally unordered rules are semantically most elegant, because an algebraic semantics can be given by interpreting the rewrite rules as equations. However, especially for human programmers, it is very time consuming to produce TRSs with the required operational behaviour. In contrast, totally ordered rules have a predictable behaviour, but the semantics of TRSs with a total order tends to be very operational in character.

In the context of innermost rewriting, syntactic specificity ordering is semantically unproblematic, see [BBKW89]. Nevertheless, it offers sufficient expressive power, even for human programmers, to specify the desired operational behaviour when this matters. For totally defined functions, a linear order may be enforced by introducing auxiliary functions. Thus, the semantic complication is introduced only where it is really required.

As a courtesy to human programmers, implementations of EPIC should provide information on rules that are neither ordered by specificity nor obviously confluent, in order to flay potentially harmfull non-determinism.

4.4 Constructing Models of the Abstract Syntax

In the previous section, we presented the semantics of EPIC in terms of a number of functions operating on EPIC's abstract syntax. In order for a tool to *use* EPIC, we need a *model* of this abstract syntax. Here, we present a way to construct such models. Note that this is not meant to be the canonical way of constructing such models; this section is only provided as an example of how one might proceed. In Section 4.4.1, we provide construction functions, and in Section 4.4.2 we use these functions to define a mapping from a concrete syntax for EPIC into the abstract syntax of EPIC.

4.4.1 Functions for the construction of EPIC

We introduce the following functions, all prefixed with `mk_` as mnemonic for the fact that these functions are used to *make* EPIC constructs from smaller parts. As in the abstract syntax, we will consider the sort `Id` as a parameter, i.e., we assume some external way to obtain identifiers. We also assume a function `mk_id` to construct an `Id` from a literal text (represented as a quoted string).

Note that we use terms to represent the sorts occurring in types, and properties such as ‘free’ and ‘external’. Only the *number* of sorts in a type is significant to EPIC, but because terms are available anyway, representing sorts by terms introduces no new notions, and leaves freedom for interpretation by external tools. Similarly, the only properties currently significant to EPIC are ‘free’ and ‘external’, but external tools may freely add and interpret properties represented by terms.

The signature of the construction functions is as follows:

```
mk_null   :                       -> Sq
mk_var    : Id                    -> Term
mk_type   : Id # Sq_t # Term      -> Type
mk_attrs  : Sq_t # Type           -> Type
mk_indx_t : Term # Sq_t           -> Sq_t
mk_term   : Id # Sq_t             -> Sq_t
mk_rule   : Term # Term           -> Rule
mk_indx_r : Rule # Sq_r           -> Sq_r
mk_indx_f : Type # Sq_f           -> Sq_f
mk_mod    : Sq_t # Sq_r           -> Mod
mk_indx_m : Mod # Sq_m            -> Sq_m
mk_prog   : Sq_m                  -> Prog
```

In order to define the semantics of properties, we introduce a predicate has, which decides whether a sequence of terms contains a term representing a certain property:

```
has: Term # Sq_t predicate
    ¬null(i) ∧ at(i) = t ⟹ has(t, i)
    ¬null(i) ∧ has(t, adv(i)) ⟹ has(t, i)
```

The semantics of the construction functions is obtained by requiring the following relationships between construction and abstract syntax functions:

```
    null(mk_null)
    at(mk_indx_t(T, Ts)) = T
    at(mk_indx_r(R, Rs)) = R
    at(mk_indx_f(F, Fs)) = F
    adv(mk_indx_t(T, Ts)) = Ts
    adv(mk_indx_r(R, Rs)) = Rs
    adv(mk_indx_f(F, Fs)) = Fs
    subs_m(mk_prog(Ms)) = Ms
    subs_f(mk_mod(Fs, Rs)) = Fs
    name(mk_attrs(mk_type(N, Ts), As)) = N
    arity(mk_attrs(mk_type(N, Ts), As)) = arity(Ts)
    free(mk_attrs(mk_type(N, Ts), As))
    = has(mk_term(mk_id("free"), mk_null), As)
    external(mk_attrs(mk_type(N, Ts), As))
    = has(mk_term(mk_id("external"), mk_null), As)
```

$$\begin{array}{l}
\mathtt{lhs}(\mathtt{mk\_rule}(L, R)) = L \\
\mathtt{rhs}(\mathtt{mk\_rule}(L, R)) = R \\
\mathtt{subs_t}(\mathtt{mk\_term}(F, Ts)) = Ts \\
\mathtt{ofs}(\mathtt{mk\_term}(F, Ts)) = F \\
\mathtt{is\_var}(\mathtt{mk\_var}(I))
\end{array}$$

4.4.2 A Concrete Syntax to model the Abstract Syntax

For human consumption and production of EPIC, a concrete syntax is needed. Only two things are relevant about this syntax; it should be readable by humans, and a mapping into the abstract syntax of EPIC should exist. Below, we first present the concrete syntax in BNF, and then its mapping into EPIC.

```
Spec     ::= Mod Spec | ϵ
Mod      ::= "module" Term "types" Types "rules" Rules
Types    ::= Type ";" Types | ϵ
Type     ::= FunId ":" HT "->" Term Props
Props    ::= "{" Term Terms "}" | ϵ
HT       ::= Term HTerms | ϵ
HTerms   ::= "#" Term HTerms | ϵ
Terms    ::= "," Term Terms | ϵ
Term     ::= VarId | FunId | FunId "(" Term Terms ")"
Rules    ::= Rule ";" Rules | ϵ
Rule     ::= Term "=" Term
FunId    ::= [a-z][-_A-Za-z0-9']* |  — identifiers start with lowercase
             """ Qchar* """        |  — quoted strings are identifiers
             "'"[!-~]              |  — all printable characters
             "\"[0-2][0-9][0-9]       — all characters; decimally coded
VarId    ::= [A-Z][-_A-Za-z0-9']*
Qchar    ::= "\"" | "\\" | ~[\"]   |  backslash and quote are escaped
```

The construction of EPIC abstract syntax from texts in this concrete syntax is straightforward, using the functions introduced in the previous section. We specify for every nonterminal N (such as `Spec`, `Mod`, `Types`, etc.) a mapping $[\cdot]_N : \mathit{Text} \rightarrow N$. We use an italic font to represent variable parts of the text, and a roman font to represent the fixed parts. Note that we assume a function `mk_id`, which is used to turn literal text into an element of the parameter sort `Id`.

$$
\begin{aligned}
[\epsilon]_{\texttt{Spec}} &= \texttt{mk\_prog}(\texttt{mk\_null})\\
[MS]_{\texttt{Spec}} &= \texttt{mk\_prog}(\texttt{mk\_indx\_m}([M]_{\texttt{Mod}}, \texttt{subs}_{\texttt{m}}([S]_{\texttt{Spec}})))\\
[\text{module } I \text{ types } T \text{ rules } R]_{\texttt{Mod}} &= \texttt{mk\_mod}([T]_{\texttt{Types}}, [R]_{\texttt{Rules}})\\
[\epsilon]_{\texttt{Types}} &= \texttt{mk\_null}\\
[T;R]_{\texttt{Types}} &= \texttt{mk\_indx\_t}([T]_{\texttt{Type}}, [R]_{\texttt{Types}})\\
[F{:}S \rightarrow T\ P]_{\texttt{Type}} &= \texttt{mk\_attrs}(\texttt{mk\_type}(\texttt{mk\_id}(F), [S]_{\texttt{HT}}, [T]_{\texttt{Term}}), [P]_{\texttt{Props}})\\
[\epsilon]_{\texttt{Props}} &= \texttt{mk\_null}\\
[\{TR\}]_{\texttt{Props}} &= \texttt{mk\_indx\_t}([T]_{\texttt{Term}}, [R]_{\texttt{Terms}})\\
[\epsilon]_{\texttt{HT}} &= \texttt{mk\_null}\\
[TR]_{\texttt{HT}} &= \texttt{mk\_indx\_t}([T]_{\texttt{Term}}, [R]_{\texttt{HTerms}})\\
[\epsilon]_{\texttt{HTerms}} &= \texttt{mk\_null}\\
[\#TR]_{\texttt{HTerms}} &= \texttt{mk\_indx\_t}([T]_{\texttt{Term}}, [R]_{\texttt{HTerms}})\\
[\epsilon]_{\texttt{Terms}} &= \texttt{mk\_null}\\
[,TR]_{\texttt{Terms}} &= \texttt{mk\_indx\_t}([T]_{\texttt{Term}}, [R]_{\texttt{Terms}})\\
[V]_{\texttt{Term}} &= \texttt{mk\_var}(\texttt{mk\_id}(V))\\
[F]_{\texttt{Term}} &= \texttt{mk\_term}(\texttt{mk\_id}(F), \texttt{mk\_null})\\
[F(TR)]_{\texttt{Term}} &= \texttt{mk\_term}(\texttt{mk\_id}(F), \texttt{mk\_indx\_t}([T]_{\texttt{Term}}, [R]_{\texttt{Terms}}))\\
[\epsilon]_{\texttt{Rules}} &= \texttt{mk\_null}\\
[Q;R]_{\texttt{Rules}} &= \texttt{mk\_indx\_r}([Q]_{\texttt{Rule}}, [R]_{\texttt{Rules}})\\
[S{=}T]_{\texttt{Rule}} &= \texttt{mk\_rule}([S]_{\texttt{Term}}, [T]_{\texttt{Term}})
\end{aligned}
$$

4.4.3 Conclusions

We have formally defined abstract syntax, operational semantics, and one of many possible concrete syntaxes of EPIC, a language based on term rewriting.

In EPIC, rules are *partially* ordered. Semantically, this is a complication with respect to unordered rules. Operationally, this gives more expressive power than unordered rules, without introducing the semantic complications of *totally ordered* rules as found in functional programming languages.

Our presentation of the abstract syntax is more abstract than usual. Instead of specifying the concrete representation of the abstract syntax, we only specify access-functions. On the one hand, this makes it easier to extend EPIC without rewriting all the code of existing tools for EPIC. On the other hand, the high level of abstractness is hard to grasp.

Chapter 5

An EPIC compiler in EPIC

> Now that we have introduced Minimal Term Rewriting Systems (MTRSs), which can be interpreted as the language of an Abstract Rewriting Machine (ARM), an executable specification of the compilation of Term Rewriting Systems (TRSs) into MTRSs, and a language for expressing TRSs (EPIC) in Chapters 3 and 4, we now complete the cycle by expressing the compiler from TRSs into MTRSs in EPIC itself.
>
> The main reason for writing the EPIC compiler in EPIC itself is portability. Additionally, expressing the compiler in its own language provides information on the usability of EPIC as a compiler construction language.
>
> The EPIC compiler is a so-called *literate* program; from a single file, both code and documentation are generated.

In order to be of any use, an EPIC program must be compiled into an executable form. In this chapter, we discuss a compiler from the programming language EPIC into Minimal Term Rewriting Systems, which is written in EPIC itself, and which uses the techniques presented in Chapter 3. Using a trivial interpretation, MTRSs can be executed on the Abstract Rewriting Machine (see Chapter 3).

The compiler is a so-called *literate program*, meaning that both program and documentation are derived from a single source, using the literate programming tool *noweb*, developed by Norman Ramsey [Ram92].

The decision to write the EPIC compiler in EPIC itself, as well as the global structure of the compiler merit some discussion. In Section 5.1.1, we discuss the requirements for the EPIC compiler, in Section 5.1.2 we mention the relative merits of EPIC and C as implementation languages, in Section 5.1.3 we deal with the lack of conditional constructs in EPIC, in Section 5.1.3 we consider the problems caused by the requirement of *representation independence*, and in Section 5.1.4, we give an overview of the compilation process. The actual code of the compiler is given in Appendix A.

5.1 Design of the Compiler

5.1.1 Requirements on a compiler for EPIC

EPIC is primarily meant as a target language for the compilation of algebraic, functional and specification languages such as ASF+SDF [BHK89a], OBJ [GWM+92], Atlas [HM93], Multi-Level Specifications [Vis96], Standard ML [Mil84], Haskell [HJW+92], etcetera. A necessary precondition for success as a target language is *extreme* portability. Dependencies on programming languages or tools, or on operating systems, are dangerous because they limit the portability of the intended source languages.

Portability limits the choice of implementation language for the EPIC compiler to either C, one of the most widely available programming languages, or EPIC itself.

Somewhat less important, but also vital, is the speed of the EPIC compiler. Prospective users of EPIC as a target language are faced with the choice between generating EPIC, or maintaining both their own abstract machine and a number of back-ends producing machine code for the most important platforms. A compiler using the EPIC compiler as a back-end can never be faster than the EPIC compiler itself.

As is discussed in Section 5.1.4, the requirement of speed implies some compromises with respect to the functional structure of the compiler.

In order to allow for future extensions of EPIC, the EPIC compiler should be *representation independent* with respect to the abstract syntax of EPIC. That is, rather than by concrete structures, access to and construction of abstract syntax nodes should proceed through access functions.

5.1.2 Relative merits of C and EPIC

As a rule of thumb, C programs execute very fast because C is close to machine language, but there are two exceptions to this rule. Firstly, most C compilers do not deal well with (tail) recursive function calls, and secondly, the memory management of dynamic data structures consisting of small elements may take exorbitant amounts of time. Both programming techniques do not occur frequently in prototypical C applications, but they do appear in a natural expression of the compilation techniques presented here. ARM, however, is designed to handle recursive functions and allocation of small structures well. This means that a priori (and disregarding the fact that the current implementation of ARM is interpreted), it is hard to decide between EPIC and C with respect to speed.

With respect to expressivity, C offers a nice supply of control structures, built-in integer arithmetic, built-in characters, arrays and strings, and side-effects. EPIC only offers innermost rewriting with syntactic specificity ordering of user-defined functions, and predefined character and string functions. A priori, rewriting is more appropriate for expressing a compilation process (rewriting a source expression into a target expression), but the requirement of representation independence prohibits extensive use of pattern matching (see Section 5.1.3 below). The lack of control structures other than pattern matching can be repaired to a certain extent (see Section 5.1.3), but not completely. Furthermore, the lack of side-effects is certainly felt. Therefore, with respect to expressivity, one would choose C as implementation language.

A notoriously difficult problem in C is memory management for dynamic data structures. C code that mainly uses such structures is cluttered with statements for allocation and release, failure to release all allocated space leads to so-called *memory leaks*, and releasing allocated space too early leads to malfunctions that can be particularly hard to track down. In contrast, the memory management for EPIC is completely automatic.

Even though many debugging tools are available, finding errors in C programs is far from easy, especially because errors usually result in (eventual) abortion of execution. An error in an EPIC program either results in an unintended normal form, or in nontermination. Errors of the first kind are usually found easily because EPIC is unconditional (see also section 5.1.3). Errors of the second kind are much harder to find, though observation of an execution trace usually indicates the error immediately. Of course, the lack of side-effects makes errors in EPIC much easier to localize.

The compilers and other tools for C are much more mature than the tooling around EPIC, which again makes C preferable as implementation language.

The decisive argument to choose for EPIC as implementation language is the fact that the compiler is a non-trivial program that can be used to assess the strengths and weaknesses of EPIC itself. And even though EPIC is primarily meant as a target language, we were of course inspired by the following quote of Niklaus Wirth:

> Use your own tools. If *you* do not use them, why would anybody else?

The future will learn how successful EPIC will be as a target language, but the preliminary conclusion with respect to EPIC as a systems programming language seems to be:

> *Improve* your own tools. If *you* do not improve them, why would anybody else?

5.1.3 Problematic aspects of representation independence

Pattern matching with specificity ordering provides most of the expressive power of EPIC. Unfortunately, the requirement of representation independence prohibits the use of pattern matching to a large extent, as is illustrated by the following example (see also [Wad87] and [BC93]). Given a concrete representation of sequences by the functions `nil` and `cons`, we can define the map of some function `f` over the sequence as follows:

```
mapf(nil) = nil;
mapf(cons(X,Xs)) = cons(f(X),mapf(Xs));
```

Representation independence requires that we do not use the fact that lists are constructed by `nil` and `cons`. Instead, we define a function `null` to determine whether a list is empty, and on the domain of non-empty lists, access functions `at` ("head") and `adv` (short for *adv*ance, also known as "tail") for obtaining the head and the tail of a list, respectively. Furthermore, we define functions `mk_null` and `mk_sq` for constructing empty and non-empty sequences, respectively.

```
null(nil) = yes;
null(cons(X,Xs)) = no;

at(cons(X,Xs)) = X;

adv(cons(X,Xs)) = Xs;

mk_null = nil;
mk_sq(X,Xs) = cons(X,Xs);
```

Now, using an auxiliary function `mapf2`, we can define `mapf` as follows:

```
mapf(L) = mapf2(null(L),L);

mapf2(yes,L) = mk_null;
mapf2(no,L) = mk_sq(f(at(L)),mapf(adv(L)));
```

The advantage of this approach is that we can now extend the datatype of sequences, without having to change the code for `mapf` or `mapf2`, or indeed, any other code using sequences.

For example, we can add annotations by adding the following rules:

```
has_annotation(annotate(A,X)) = yes;
has_annotation(X) = no;

get_annotation(annotate(A,X)) = A;

null(annotate(A,X)) = null(X);
at(annotate(A,X)) = at(X);
adv(annotate(A,X)) = adv(X);
```

Making a program representation independent causes three problems.

Firstly, the functions `at` and `adv` are partial. However, they can be made total by using three- or four-valued logic, as discussed in [BS96]. This is an important technique, but we will not discuss it any further here.

Secondly, we now need 6 functions and 6 rules to define sequences, whereas 2 constructors sufficed previously. Assuming that sequences are used many times in the rest of the program, this is not a big price to pay: all rules in which `nil` or `cons` appears in the left-hand-side (LHS) would be duplicated in order to make the old program deal with annotations of sequences.

Thirdly, we now need 2 functions and 3 rules to define `mapf`, and it is clear that this definition of `mapf` is neither as natural nor as readable as the first one. In the next section, we provide a partial solution to this problem.

Introducing a Conditional Construct by a Source-to-Source Transformation

To a certain extent, the problems sketched above can be repaired using *conditions*:

```
null(L) = yes
=============
mapf(L) = mk_null

null(L)       = no
f(at(L))      = First
mapf(adv(L))  = Rest
===========================
mapf(L) = mk_sq(First,Rest)
```

Even though the meaning of this program is intuitively clear, the semantics of conditional rewriting is problematic. Without discussing them in detail, we mention the following issues:

- The operational semantics of conditions, i.e., in what order are conditions evaluated.
- Conditional rewriting in the context of non-confluent or weakly terminating systems.
- The definition of specificity ordering for conditional rewriting.
- The semantics of negative conditions.
- Debugging of programs with conditions is significantly more difficult than debugging unconditional programs (see below).

To resolve these issues, many suitable, but incompatible, proposals have been made in the literature, so adding conditions to EPIC would involve a number of choices. Evidently, such conditional constructs in EPIC can only be used directly for the translation of source languages where the same choices have been made. For other source languages, the semantics chosen in EPIC would be less than useless. In the worst case, it could become a source of hard-to-find errors.

As mentioned above, conditions should also be regarded with suspicion during the development of a program. When one of the conditions for a certain rule fails (e.g., because a case was forgotten for some function definition), the term that matches the LHS of the rule is not rewritten. This means that the normal form obtained does not provide any information concerning which of the conditions failed, nor why it failed. To some extent, this is the same problem that makes Prolog programming difficult: failure to prove some intermediate result in the proof of a complicated query leads to the uninformative result '`no`'.

To satisfy the need for conditions, without paying the price of an extended semantics, we define syntactic sugar for a simple source-to-source transformation on EPIC programs.

This syntax is not powerful enough to express general conditional rewriting, but it does help to address the problems introduced by representation independence, and its (transformational) semantics is completely unproblematic.

The new syntax extends the terms (`RTERM`) that may occur as right-hand-side (RHS) of an equation (assuming `LTERM` for terms that may occur on the LHS)

```
RTERM  ::= "given" RTERM+ as CASE+ ["end"]
RTERM+ ::= RTERM | RTERM RTERM+
CASE+  ::= CASE | CASE CASE+
CASE   ::= LTERM+ : RTERM
```

The optional keyword `"end"` is meant for disambiguation of nested occurrences of this construct. Also, the number of `LTERM`s in each case should coincide with the number of `RTERM`s between the corresponding `given` and `as`.

We may use this syntax to define `mapf` as follows:

```
mapf(L) =
  given null(L) as
    yes: mk_null
    no : given f(at(L)),mapf(adv(L)) as First,Rest :
           mk_sq(First,Rest);
```

The semantics is as follows. A RHS in which the new syntax occurs is replaced by a term built from an auxiliary function with a name derived from the name of the outermost function symbol (OFS) of the LHS, with as arguments the terms occuring in the premise, and all variables occuring in the LHS. For every case, a rule is introduced for the new function. For our example, elimination of the new syntax yields:

```
mapf(L) = "g1_mapf"(null(L),L);

"g1_mapf"(yes,L) = mk_null;
"g1_mapf"(no,L) = "g2_g1_mapf"(f(at(L)),mapf(adv(L)),L);

"g2_g1_mapf"(First,Rest,L) = mk_sq(First,Rest);
```

We see the following advantages of this approach:

- The transformation is easily implemented as a separate tool producing EPIC. If this functionality is not wanted, the tool and its documentation can be ignored.
- The semantics of this construct is obvious, it follows directly from the semantics of EPIC itself. Especially with respect to specificity ordering, this construct is unproblematic.

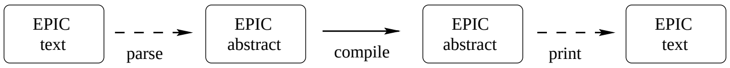


Figure 5.1: A global overview of the EPIC compiler

- Unlike real conditions as discussed above, failure of a condition does not lead to returning the original term as normal form. Instead, the normal form will contain one of the newly introduced functions, which gives a clear indication which condition failed and why it failed.

With respect to real conditions the major disadvantage is that one has to be careful with rules that have overlapping LHSs. Given two rules with overlapping LHSs, the choice between these rules is made before the 'conditions' are evaluated, so when a condition fails, reduction of the other rule is not attempted.

Finally, it should be noted that representation independence introduces another problem. Consider the following fragment:

```
compile(nil) = nil;
compile(cons(rule(ap(F,Largs),ap(H,Rargs)),Rules) = ...
```

Even with the use of our conditional construct, an equivalent fragment would be:

```
compile(Rules) =
  given null(Rules) as
    yes : Rules
    no  : given at(Rules),adv(Rules) as Rule,Rules :
          given lhs(Rule),rhs(Rule) as Lhs,Rhs :
          given ofs(Lhs),ofs(Rhs) as F,H :
          given subs(Lhs),subs(Rhs) as Largs,Rargs : ...,
```

which is unattractive, because it is much longer than the fragment above. This problem is not inherent in EPIC, but a direct consequence of representation independence. A technique is needed that allows one to write the first fragment where the second is meant. Research in this direction was done by [Wad87], and [BC93], but the results are not satisfactory yet.

5.1.4 Overview of the compiler

In figure 5.1, we give a global overview of the structure of the compiler. To compile an EPIC text, the text is first parsed into an abstract syntax representation, then it is compiled into an MTRS, again in abstract syntax representation, which is finally printed as text again. The final printing phase could of course be replaced by a phase producing ARM code directly from the MTRS, and the initial parsing phase could be omitted in case the EPIC abstract syntax is delivered by some other tool, such as a compiler for a

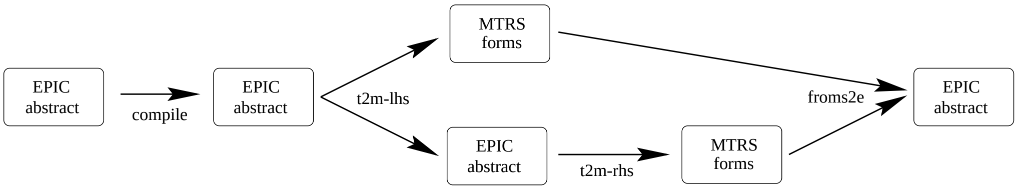


Figure 5.2: The compilation process

higher-level language. This is the reason that we will discuss neither the parser nor the printer.

Zooming in a little bit, figure 5.2 gives an overview of the compilation process itself. First, an EPIC program represented in EPIC abstract syntax is transformed by `complete` (see Appendix A.7.1) into a simply complete TRS, again represented in EPIC abstract syntax.

Then, this TRS is transformed by `t2m-lhs` (see Appendix A.7.2) to make the LHSs conformant with MTRS rules. This process yields rules of two types; rules that are already in MTRS form, and rules that are not. In order to save space, and to prevent costly recomputations, the MTRS rules are represented differently from the non-MTRS rules (see Section A.6). Unfortunately, this compromises the functional structure of the compiler. A similar source of clutter is the fact that the 'locus annotations', i.e. annotations indicating how the arguments are divided over the T and A stacks, are computed on the fly. At the cost of yet another analysis, this could have been put in a separate phase.

In a subsequent phase, `t2m-rhs` (see Appendix A.7.3), the non-MTRS rules are transformed into MTRS rules. Again, this phase is cluttered by the computation of locus annotations.

Finally, `forms2e` (see Appendix A.6) is used to convert the MTRS forms back into EPIC abstract syntax.

5.2 Conclusions

We have written a compiler for EPIC in EPIC itself. Even though it handles pattern matching and variable allocation, the size of the compiler is moderate. The code is independent of the representation of the abstract syntax of EPIC, which will doubtlessly simplify the implementation of extensions to EPIC.

Unfortunately, representation independence causes a laborious style of programming, because the use of pattern matching and specificity ordering is severely limited. We conclude from this that a technique enabling the use of pattern matching without introducing a dependence on a particular representation is wanted very badly. Initial investigations into this problem have been done ([Wad87, BC93]), but the results are still unsatisfactory.

By introducing a conditional construct, we have improved the expressive power of EPIC. In order to avoid the pitfalls of general conditional rewriting, the semantics of this construct is defined by a simple source-to-source transformation.

Apart from ensuring portability, the main reason to write the EPIC compiler in EPIC itself was to assess the strengths and weaknesses of EPIC as a language for compiler construction. Unfortunately, the requirement of representation independence proved to be the main source of problems arising during the construction of this compiler, which makes it hard to draw definitive conclusions on the usability of EPIC as a compiler building language.

Chapter 6

Lazy Rewriting on Eager Machinery

Innermost rewriting, as implemented by the tools presented in the previous chapters, is not always a desirable strategy.

Here, we define *Lazy Term Rewriting* and show that it can be realized by local adaptations of an *eager* implementation of conventional term rewriting systems. The overhead of lazy evaluation is only incurred when lazy evaluation is actually performed.

Our method is modelled by a transformation of term rewriting systems, which concisely expresses the intricate interaction between pattern matching and lazy evaluation. The method easily extends to term *graph* rewriting. We consider only left-linear, confluent term rewriting systems, but we do not require orthogonality.

6.1 Introduction

It is well-known that outermost rewriting strategies often have a better termination behaviour than innermost rewriting strategies [O'D77, HL91a]. *Eager evaluation* (a conventional name for the use of innermost strategies) can be implemented much more efficiently, however. We propose to solve this dilemma by transforming a term rewriting system (TRS) in such a way that the termination behaviour of innermost rewriting is improved. At the core of our transformation are established ideas of Ingermann [Ing61] and Plotkin [Plo75].

The TRS defined in Figure 1 illustrates the bad termination behaviour of innermost rewriting.

$$\begin{array}{rcll} inf(x) & \rightarrow & cons(x, inf(s(x))) & (1) \\ nth(0, cons(x,y)) & \rightarrow & x & (2) \\ nth(s(x), cons(y,z)) & \rightarrow & nth(x,z) & (3) \end{array}$$

Figure 1

The term $\mathit{nth}(0, \mathit{inf}(0))$ has an infinite reduction sequence, since $\mathit{inf}(0) \overset{(1)}{\to} \mathit{cons}(0, \mathit{inf}(s(0))))$ $\overset{(1)}{\to} \mathit{cons}(0, \mathit{cons}(s(0), \mathit{inf}(s(s(0))))) \overset{(1)}{\to} \ldots$. This can be avoided by applying rule (1) only once, before applying rule (2): $\mathit{nth}(0, \mathit{inf}(0)) \overset{(1)}{\to} \mathit{nth}(0, \mathit{cons}(0, \mathit{inf}(s(0)))) \overset{(2)}{\to} 0$. By postponing some reductions, outermost rewriting may succeed in avoiding them altogether. 'Optimal avoidance' is studied, amongst others, in [O'D77, Fie90, HL91a, Mar92]. However, this work crucially depends on (weak) orthogonality (no overlap of patterns or rules) of TRSs. Our method applies to arbitrary TRSs, and it may even help for terms which do not terminate under any strategy, such as $\mathit{cons}(0, \mathit{inf}(0))$.

In *lazy functional programming languages* (LFPLs) [PvE93], this problem is addressed by a technique called *lazy evaluation.* This concept needs a generalization in order to be applicable to TRSs, because the textual order of rules and patterns is significant in LFPLs, but not in TRSs. Before turning to our generalization, which we call *lazy rewriting*, we will briefly discuss lazy evaluation.

During lazy evaluation, guided by textual order, non-outermost redexes are contracted in order to establish that the outermost function symbol will never become part of a redex. The resulting term is said to be in Weak Head Normal Form (WHNF). E.g., the term $\mathit{cons}(0, \mathit{inf}(0)))$ in the example above is in WHNF. For this term, the (implicit) output routine of an LFPL first produces $\mathit{cons}(0,$ on output, before recursively reducing the argument to WHNF. The term $\mathit{cons}(0, \mathit{inf}(0))$ still leads to an infinite computation, but 'useful' output is produced during this computation. Because rewriting of outermost redexes is expensive, it is usually avoided as much as possible. Arguments that can be rewritten eagerly without affecting termination behaviour, are called *strict.* Strictness analysis (initiated by Mycroft, [Myc80]) attempts to identify these arguments statically.

$$\begin{array}{rcll}
\mathit{inf}(x) & \to & \mathit{cons}(x, \mathit{thunk}(x)) & (1) \\
\mathit{inst}(\mathit{thunk}(x)) & \to & \mathit{inf}(s(x)) & (2) \\
\mathit{nth}(0, \mathit{cons}(x, y)) & \to & x & (3) \\
\mathit{nth}(s(x), \mathit{cons}(y, z)) & \to & \mathit{nth}(x, \mathit{inst}(z)) & (4)
\end{array}$$

Figure 2

Now consider the TRS in Figure 2, which differs only slightly from the one in Figure 1. The term $\mathit{nth}(0, \mathit{inf}(0))$ still rewrites to 0, but there are no infinite reduction sequences. This example illustrates that only minor changes are needed to achieve the desired effect, and that these changes can be made local to "lazy positions" (cf. the second argument of *cons*). To some extent, this explains the success of strictness analysis. In many cases, only a few positions need a lazy treatment in order to preserve termination.

The example also demonstrates the common observation that a good implementation of a lazy language spends most time in "eager mode". Given the locality of the changes above, it is worthwhile to investigate how an *eager* implementation can be adapted to do some *lazy* evaluation, rather than adapting a *lazy* implementation to do (a lot of) *eager* evaluation.

We use laziness annotations to indicate argument positions where rewriting should be postponed if possible. These annotations could be provided by the programmer or by a strictness analyzer. In the latter case, all arguments that are not found to be strict, will get the annotation *lazy*, and the reductions performed by our implementation will correspond closely to the reductions performed by an implementation of an LFPL using the same strictness analyzer.

Even though Figure 2 is a simplified version of the result of our transformation, applied to the TRS of Figure 1 (with only the second argument of *cons* annotated with *lazy*), it exhibits a peculiarity of our scheme. The term *inf*(0) rewrites to the normal form *cons*(0, *thunk*(0)), which is not a term in the original signature. However, the term *thunk*(0) (called a "thunk" after Ingermann [Ing61]) represents a (possibly infinite) term in the original system, which can be further approximated by repeatedly replacing terms *thunk*(x) by the normal form of *inst*(*thunk*(x)). Our lazy normal forms (LNFs) generalize WHNF, and the approximation process corresponds to what is done by the output routine in an implementation of an LPFL. We do not assume such an output routine, because leaving the thunk in place offers the possibility of preventing uninteresting work, and yields a larger class of terminating systems.

So far, we have avoided mentioning one important aspect of lazy evaluation in LFPLs: the use of graphs instead of terms in order to prevent multiple evaluations of the same term. Conventionally, postponing evaluation without sharing results is called *call by name*, whereas true lazy evaluation is alternatively called *call by need.* In our presentation, we will first concentrate on call by name, and then show that by providing an alternative implementation for one of the auxiliary functions, an implementation of *call by need* is obtained.

We give our definition of *lazy* term rewriting in Section 6.2, a complete version of the transformation sketched above in Section 6.3, and some remarks on a realistic implementation in Section 6.4. We end with a discussion of related work and conclusions.

6.2 Lazy Term Rewriting

We define *lazy term rewriting* as term rewriting (see Section ??) with a restriction on the (one-step) rewrite relation. First, we define *lazy* signatures, which make a distinction between *eager* argument positions and *lazy* argument positions.

The choice to annotate the *arguments* rather than the *function symbols* themselves is not only motivated by compatibility with LFPLs, but has two additional advantages. First, if functions are annotated, we must expect thunks at every argument position, thus losing the locality of our transformation. Second, for functions such as `if(Bool,Exp,Exp)`, it is more natural to annotate an argument position than to annotate all function symbols that may occur there. Unfortunately, not all TRSs can be made terminating by only annotating arguments (cf. the rule $inf(x) = inf(x)$).

A lazy signature includes a predicate Λ on function symbols and natural numbers, where $\Lambda(F, i) = true$ means that the ith argument position of F $(0 \leq i \leq arity(F))$ is lazy, and $\Lambda(F, i) = false$ means that it is eager. As an abbreviation, we write $F(!, ?)$ for a function F of arity 2, the first argument of which is eager and the second argument of which is lazy.

This notion is easily extended:

Definition 31 *Lazy and eager paths in terms.*

- *For all terms t, ε is an* eager path *in t.*
- *If p is an eager path in t and $t|_p = F(t_1, \ldots, t_n)$ with $\neg\Lambda(F, i)$ for some i $(1 \leq i \leq n)$ then $p.i$ is an eager path in t.*
- *All other paths are* lazy.

In other words, a path is *eager* precisely if it passes through eager arguments only. A lazy path p in t is called *directly lazy* if $p = p'.i$ with i a one-element path, and i lazy in $t|_{p'}$. For example, given the signature $cons(!,?)$, $\mathrm{bin}(!,!)\}$ and the terms $t_1 = cons(x, cons(y, z))$ and $t_2 = \mathrm{bin}(cons(x, y), cons(x, z))$, the paths 1 in t_1 and $1, 1.1, 2, 2.1$ in t_2 are eager; $2, 2.1, 2.2$ in t_1 and $1.2, 2.2$ in t_2 are lazy, of which only 2.1 in t_1 is not also directly lazy.

With *Lazy*(t), we will denote the prefix reduced set of lazy paths in t, and a subterm at a lazy path will be called a *lazy subterm.* By $\zeta(t)$ we will denote the term obtained from t by substitution of the special constant symbol ζ at every $p \in \mathrm{Lazy}(t)$. If p is lazy in t, then $\mathrm{I}(p)$ is the initial segment of p that belongs to $\mathrm{Lazy}(t)$.

We will say that terms t_1 and t_2 are ζ-equal, or equal *up to* ζ, when $\zeta(t_1) = \zeta(t_2)$.

Ideally, we would like to rewrite a lazy subterm at path p only if this is *needed* to establish a *needed* redex at an eager prefix e of p. Then, the termination behaviour of lazy rewriting would be at least as good as the termination behaviour of rewriting only needed redexes. Unfortunately, because we do not want to restrict ourselves to sequential systems, rewriting needed redexes is not feasible (see [HL91a]). Therefore, we give a weaker definition, which only requires that a redex at an eager prefix of p can be established by replacing lazy subterms.

The ideal of *needed* rewriting can be approximated by demanding a particular relation between the lazy subterms and their replacements. We will not try to achieve this, because the most interesting relations seem to be either undecidable or hard to implement or have such a large bias towards a particular strategy that they are unnatural as a restriction on the rewrite relation.

Instead, we try to make the restriction on the rewrite relation as weak as possible, by considering only LHSs and outermost lazy paths. The rewrite *strategy* is expected to approximate the ideal by avoiding as many rewrites at lazy paths as reasonably possible. The transformation presented in Section 6.3 implements such a strategy.

Definition 32 *Let the path p in a term t be a redex for some rule with left-hand side u. An extension $p.q$ of p is* inessential *for p with regard to u if q has an initial segment r such that $u|_r$ is a variable. The other extensions of p in t are* essential *for p with regard to u.*

An extension is essential for a redex path p if it is essential with regard to *some* left-hand side.

Definition 33 *Let $\mathcal{R}$ be a term rewriting system with laziness predicate Λ. A redex p in a term t of $\mathcal{R}$ is* free *if p is eager* or *p is an extension of an eager path e that is a redex in some $t' \equiv_\zeta t$ for which* $\mathrm{I}(p)$ *is essential. A redex that is not free is* suppressed.

We write $s \rightarrow_L t$ (s rewrites to t *lazily*) if s reduces to t by contraction of a free redex, and $s \rightarrow_S t$ if a suppressed redex is contracted. Because a suppressed is always at a lazy path, $s \rightarrow_S t$ implies that $s \equiv_\zeta t$.

The restriction $\rightarrow_L$ of the one-step rewrite relation yields an extended class of normal forms. We will call these *lazy normal forms (LNF)*. For instance, given the TRS of Figure 1, if $\Lambda(cons, 2) = true$, then $cons(0, inf(0))$ is an LNF which is not a normal form. If $\Lambda(f, i)$ is *true* for all f, i, LNF coincides with WHNF. If t is an LNF, we call $\zeta(t)$ a ζ-LNF.

Observe that whether a redex r is suppressed depends on circumstances that are not influenced by contraction of suppressed redexes. Hence:

Lemma 3 *Let p, q be suppressed redexes in s, $s \rightarrow_S t$ by contraction at q, and r a free redex in t. Then*

(a) the descendants of p in t are suppressed (in as far as they are redexes at all);

(b) r is a free redex in t, for the same rules as in s.

Lemma 4 *Suppose $t_1 \rightarrow_S^* t_2$ by a development of disjoint redexes, and $t_2 \rightarrow_L^* t_3$. Then t_1 has a lazy reduct that is ζ-equal to t_3.*

Proof. Induction on the length of the given reduction from t_2 to t_3. If the length is 0, then $t_1 \equiv_\zeta t_2 \equiv_\zeta t_3$. Now let $t_2 \rightarrow_L t_4$ be the first step of the reduction to t_3 . The redex contracted there already exists in t_1, by lemma 1(b). Say that contraction of this redex in t_1 leads to t_4'. Then by lemma 1(b) once more, $t_1 \rightarrow_L t_4'$. This reduction does not disturb the suppressed redexes developed in $t_1 \rightarrow_S^* t_2$; it just may eliminate or multiply them. Moreover, the descendants of these redexes are disjoint. Development of these descendants leads to t_4. First develop the descendants that have got free ($t_4' \rightarrow_L^* t_5$), and then what remains suppressed. (By lemma 1(a), in the latter part of the development no free redexes are created.) By induction hypothesis there exists $t_3' \equiv_\zeta t_3$ such that $t_5 \rightarrow_L^* t_3'$. Hence $t_1 \rightarrow_L^* t_3'$.

Theorem 1. To every reduct of a given term there exists a ζ-equal lazy reduct.

Proof. Induction on the length of the given reduction $s \rightarrow^* t$. Suppose we have $s \rightarrow s' \rightarrow^* t$, and a lazy reduction $s' \rightarrow_L^* t' \equiv_\zeta t$. If $s \rightarrow_L s'$, t' is a lazy reduct of s as well. If $s \rightarrow_S s'$, s has a lazy reduct $t'' \equiv_\zeta t'$.

In particular, since normal forms are lazy normal forms and a term ζ-equal to an LNF is itself an LNF:

Corollary 1. To every NF of a given term there exists a ζ-equal LNF.

Of course, reducts of distinct terms may be ζ-equal.

Corollary 2. If two terms have ζ-equal reducts, they have ζ-equal lazy reducts.

From the fact that an arbitrary number of "irrelevant" rewrite steps can in general be performed before the rewrite that turns a term into an LNF, it follows that (local) confluence is not preserved. However, given the fact that we are only really interested in $\zeta(n)$ for any LNF n, it is fair to consider only (local) ζ-*confluence*:

Definition 34 *A TRS R is (locally)* ζ-confluent *if for every t_1, t_2 and t_3, if $t_1 \rightarrow^{(*)} t_2$ and $t_1 \rightarrow^{(*)} t_3$ then there are terms t_4, t_5, such that $t_2 \xrightarrow{*} t_4, t_3 \xrightarrow{*} t_5$ and $\zeta(t_4) = \zeta(t_5)$.*

Corollary 3. Lazy rewriting preserves ζ-confluence and local ζ-confluence.

6.3 A transformation to achieve laziness

Here, we will specify a transformation $\mathcal{L}$ from TRSs to TRSs and a transformation $\mathcal{T}$ from terms to terms, such that when $\mathcal{T}(t)$ is rewritten by an innermost strategy in $\mathcal{L}(R)$ to a normal form n, then $\zeta(n)$ is the ζ-LNF of t with respect to R. The transformed system avoids rewriting lazy subterms to a large extent. Basically, the transformation $\mathcal{T}$ replaces all lazy subterms of an input term by irreducible encodings (*thunks*), and $\mathcal{L}$ supplies rules for "unthunking" both input thunks and thunks that encode right-hand sides. Furthermore, $\mathcal{L}$ ensures that

- Lazy subterms of right-hand sides are thunked. Thunks are irreducible because they are built from a constructor symbol and (already normalized) subterms of the left-hand side, so reduction at lazy paths is blocked temporarily.
- When a subterm (matched to a variable) is moved from a lazy *lhs* path into an eager *rhs* path, it is unthunked, so thunks only occur at lazy paths.
- A lazy argument is unthunked before a match overlapping with it is rejected.

We start with some definitions. A *thunk* is a term with a special function symbol δ at the top, a name of a pattern (p) as first argument, and a tuple of terms (denoted by $\mathbf{vec}_n(t_1, \ldots, t_n)$) as second argument:

$$\delta(p, \mathbf{vec}_n(t_1, \ldots, t_n))$$

Given a rule $s \rightarrow t$, we call a variable *migrant* if it occurs at a *directly* lazy path in s and at some eager path in t. Because we want to keep the effect of our transformation local, rules must be added that 'unthunk' migrant variables.

6.3.1 The transformation $\mathcal{L}$

$\mathcal{L}$ takes a TRS (Σ, R), and transforms it into a system $(\Sigma \cup N \cup A, RG \cup RI \cup R')$. In the transformed system,

- N is a set of function symbols that do not occur in Σ (they are used in thunks as names of patterns). There is a set $T \subset N$ of "tokens", such that for every function symbol f in Σ, we have a unique $t_f \in T$,
- A is a set of "administrative" function symbols

$$\{\delta(!,!), \delta?(!), \mathbf{inst}(!), \pi(!,!), \mathbf{true}\} \cup \{\mathbf{vec}_{mn}(l_1, \ldots, l_n) | m, n \in N, l_i \in \{?, !\}\},$$

where δ will be used as the top symbol of a thunk, $\delta?$ is a predicate that recognizes thunks, a function $\mathbf{vec}_{mn}(l_1 \ldots l_n)$ is used to "pack" n variables in a thunk (the subscript m is used to differentiate between laziness annotations for the same arity, usually, m can be omitted). Finally, π is a projection function that makes implementation of *graph rewriting* easy, which will be discussed in Section 6.4.1.

- RG contains the general rules defining the projection π and the thunk-recognizer $\delta?$:

$$\pi(x, y) \to y$$

$$\delta?(\delta(x, y)) \to \mathbf{true}$$

- RI contains the rules describing selective unthunking of input terms. For every f with arity n, of which k are eager positions (with indices $e_1, \ldots, e_k$), RI contains the rules (with $f_{ci} \in N$):

$$\begin{array}{rcll}
\mathbf{inst}(\delta(t_f, \mathbf{vec}_n(x_1, \ldots, x_n))) & \to & f_{c1}(\delta?(x_{e_1}), x_1, \ldots, x_n) & (6.1)\\
f_{c1}(\mathbf{true}, x_1, \ldots, x_n) & \to & f_{c2}(\delta?(x_{e_2}), x_1, \ldots, \mathbf{inst}(x_{e_1}), \ldots, x_n) & (6.2)\\
f_{c1}(\delta?(y), x_{e_1}, \ldots, x_n) & \to & f_{c2}(\delta?(x_{e_2}), x_1, \ldots, x_{e_1}, \ldots, x_n) & (6.3)\\
 & \ldots & & \\
f_{ck}(\mathbf{true}, x_1, \ldots, x_n) & \to & f(x_1, \ldots, \mathbf{inst}(x_{e_k}), \ldots, x_n) & (6.4)\\
f_{ck}(\delta?(y), x_1, \ldots, x_n) & \to & f(x_1, \ldots, x_{e_k}, \ldots, x_n) & (6.5)
\end{array}$$

Here, (6.1) starts the instantiation of a delayed term with function symbol f, (6.2,6.4) handle the case that an argument (x_{e_1} and x_{e_k}, respectively) is still thunked and (6.3,6.5) handle the case that an argument is already unthunked. Note that the distinction between thunked and unthunked arguments relies on the partial function $\delta?$ being evaluated eagerly.

- The rules in R' are obtained by applying the three transformations below (RHS for thunk introduction, LR for left-right unthunking and LS for left-hand side matching) to R as follows: RHS until fixpoint, LR once for every equation in the fixpoint, LS once for every equation in the result of LR.

RHS (Thunk Introduction) This transformation is applicable to all rules $r : s \to t$ where t contains a directly lazy path p, such that $t|_p$ is neither a variable, nor a subterm already occurring in s, nor a thunk. Let $\{t_1, \ldots, t_n\}$ be the set of terms occurring in both s and $t|_p$, and prefix-reduced with respect to t, then r is *replaced* by two rules (i unique in N):

$$\begin{array}{rcl} s & \rightarrow & t[\delta(i, \mathbf{vec}_n(t_1, \ldots, t_n))]_p \\ \mathbf{inst}(\delta(i, \mathbf{vec}_n(x_1, \ldots, x_n))) & \rightarrow & \pi(\delta(i, \mathbf{vec}_n(x_1, \ldots, x_n)), t[t_i/x_i]) \end{array}$$

Here $\Lambda(\mathbf{vec}_n, i) = ?$ if and only if t_i is a variable occuring at a directly lazy path in s.

LR (Migrant Thunk Elimination) This transformation applies to rules $r : s \rightarrow t$ containing *migrant* variables. Supposing $\{t_1, \ldots, t_n\}$ is a set of subterms which occur both in s and t, and which is prefix-reduced with respect to t, and let $e_1, \ldots, e_k$ be the indices of the *migrant* variables, then r is replaced by the following rules, similar in form and intent to the rules in *RI*:

$$\begin{array}{rcl} s & \rightarrow & c_{i_1}(\delta?(x_{e_1}), t_1, \ldots, t_n) \\ c_{i_1}(\mathbf{true}, x_1, \ldots, x_n) & \rightarrow & c_{i_2}(\delta?(x_{e_2}), x_1, \ldots, \mathbf{inst}(x_{e_1}), \ldots, x_n) \\ c_{i_1}(\delta?(y), x_1, \ldots, x_n) & \rightarrow & c_{i_2}(\delta?(x_{e_2}), x_1, \ldots, x_{e_1}, \ldots, x_n) \\ \ldots & & \\ c_{i_k}(\mathbf{true}, x_1, \ldots, x_n) & \rightarrow & c_{i_k+1}(x_1, \ldots, \mathbf{inst}(x_{e_k}), \ldots, x_n) \\ c_{i_k}(\delta?(y), x_1, \ldots, x_n) & \rightarrow & c_{i_k+1}(x_1, \ldots, x_{e_k}, \ldots, x_n) \\ c_{i_k+1}(x_1, \ldots, x_n) & \rightarrow & t[t_i/x_i] \end{array}$$

LS (Matching Thunk Elimination) This transformation is applicable to rules $r : s \rightarrow t$ if s contains nonvariable lazy paths. For every element $i = \{i_1, \ldots, i_n\}$ in the prefix-reduced powerset of lazy paths in s, add a rule (all n_j and v_j fresh):

$$s[\delta(n_1, v_1)]_{i_1} \ldots [\delta(n_n, v_n)]_{i_n}$$

$$\rightarrow s[\pi(\delta(n_1, v_1), \mathbf{inst}(n_1, v_1))]_{i_1} [\delta(n_2, v_2)]_{i_2} \ldots [\delta(n_n, v_n)]_{i_n}$$

6.3.2 The transformation $\mathcal{T}$

$\mathcal{T}$ thunks all non-variable lazy subterms of the original input term, by the token of their outermost function symbol and their thunked arguments.

$$\begin{array}{rcl} \mathcal{T}[f(t_1, \ldots, t_n)] & = & f(t'_1, \ldots, t'_n) \ \ (\text{where } t'_i = \mathcal{T}[t_i] \text{ iff } \neg\Lambda(f, i), \text{ otherwise } t'_i = \mathcal{T}_l[t_i]) \\ \mathcal{T}_l[f(t_1, \ldots, t_n)] & = & \delta(t_f, \mathbf{vec}_n(\mathcal{T}_l[t_1], \ldots, \mathcal{T}_l[t_n])) \\ \mathcal{T}[x] & = & \mathcal{T}_l[x] = x \end{array}$$

6.3.3 Correctness and completeness of the transformation

First, we remark that the transformation itself terminates, because every application of RHS replaces one (non-thunked) lazy argument by a thunk, and LR and LS terminate trivially.

Theorem 2 *(Correctness of $\mathcal{L}$ and $\mathcal{T}$) Given a TRS R and a term t, every step in an innermost rewriting of $\mathcal{T}(t)$ in $\mathcal{L}(R)$ is either an administrative step (checking if an argument is a thunk), or it corresponds to a legal step in $\rightarrow_L$.*

Proof Note that for all terms t, $\mathcal{T}(t)$ has only R-redexes above lazy paths, because all lazy subterms are thunked by $\mathcal{T}_l$. By RHS, all rules have been transformed into rules that put normal forms at lazy paths, and LR preserves this property. The only redexes at lazy paths are $\mathcal{L}(R)$-redexes, introduced by LS, but the conditions for application of LS imply that there is a nonvariable R-pattern overlapping with the hole in which the redex is introduced, so the condition for lazy rewriting is fulfilled ∎

Theorem 3 *(Completeness of $\mathcal{L}$ and $\mathcal{T}$) Given a ζ-confluent TRS R and a term t, any normal form t_n of $\mathcal{T}(t)$ with respect to $\mathcal{L}(R)$ is ζ-equal to any LNF t_l of t.*

Proof Because of correctness, we have that $t \overset{LR}{\rightarrow} t_n'$, where t_n' is obtained from t_n by replacing thunks with the terms they encode (so $\zeta(t_n') = \zeta(t_n)$). Because t_l is a LNF of t, we have that $t \overset{LR}{\rightarrow} t_l$. Lazy rewriting preserves ζ-confluence, so there must be a term t_l' with $\zeta(t_l) = \zeta(t_l')$ such that $t_n' \overset{LR}{\rightarrow} t_l'$. Because t_n only differs from t_n' by having thunks at lazy paths, and LS introduces rules that remove any thunk which blocks matching of an LHS at an eager path, all the rewrites in the sequence $t_n' \overset{LR}{\rightarrow} t_l'$ occur at lazy paths. Therefore, $\zeta(t_l') = \zeta(t_n')$, so $\zeta(t_l) = \zeta(t_n)$ ∎

6.4 From transformation to implementation

The transformation in Section 6.3 is useful both as a tool for experimentation, and as a concise model of an implementation of lazy rewriting. To obtain an implementation that competes with special-purpose lazy implementations such as TIM ([FW87]) or the Spineless Tagless G-machine (STG, [JS89]), some details have to be changed.

First, in order to prevent multiple reductions of the same term, the TRS should be interpreted as a *graph* rewriting system. We give details on this in Section 6.4.1.

Second, some glaring inefficiency is caused by the LS transformation. This can be overcome by simulating the effect of LS in the pattern-matching code, which is explained in Section 6.4.2.

6.4.1 Graph rewriting by adding sharing

By the following modifications, the advantages of graph rewriting are incorporated:

- In the implementation of $\mathcal{T}$, sharing should be retained.

$$\langle \mathbf{project}{\cdot}p, P, C, T, a_1{\cdot}a_2{\cdot}A, \cdots a_1 : t_1 \cdots a_2 : t_2 \cdots\rangle = \langle p, P, C, T, a_2{\cdot}A, \cdots a_1 : t_2 \cdots a_2 : t_2 \cdots\rangle$$

Figure 6.1: A fast implementation of π in ARM

- The function $\pi(!, !)$ is implemented such, that it overwrites its first argument (always a thunk) with the LNF of its second argument (always the LNF corresponding to the thunk). Note that this requires a fixed node size, or some other means to avoid overwriting smaller with bigger nodes. See Figure 6.1 for an addition to the algebraic semantics of Figure 3.4.
- If a subterm occurs both in LHS and RHS of some rule, no copy should be made. Then it follows from the construction of the transformed system, that thunks are never duplicated, so every thunk is only evaluated once.
- For cyclic graphs, the code that is generated for the construction of a right-hand side must be modified slightly. Without loss of generality, we consider a prototypical RHS $x : f(\ldots, x, \ldots)$. For this RHS, the compiler should emit code corresponding to $\mathbf{inst}(T)$, where T is a thunk for $f(\ldots, T, \ldots)$. Note that this requires that the "address" of a node under construction is available during the construction.

6.4.2 Optimizations

When implemented naively, our transformation has a large impact on the number of equations. A worst-case analysis shows that the maximal number of additional equations is

$$3 + n.r + n.2^l + 2s$$

where n is the number of rules, r is the maximal number of nonvariable lazy paths in a RHS, l is the maximal number of nonvariable lazy paths in a LHS, and s is the number of lazy positions in the signature. It should be noted that, measured in function symbols, the rules added by *RHS* are compensated for by a reduction in size of the original rule, and s is generally small compared to n.

Thus, the only dangerous term is the exponential term in l, caused by the powerset construction in transformation *LS*. We will illustrate both the problem and its solution with an example. Assuming we have a signature $\{a, b, i(!), t(?, ?)\}$ and a rule $i(t(a, b)) \to a$, then *LS* adds the rules

$$\begin{aligned}
i(t(\delta(p, \mathbf{vec}_0), b)) &\to i(t(\pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0))), b)) \\
i(t(\delta(p, \mathbf{vec}_0), \delta(p', \mathbf{vec}_0))) &\to i(t(\pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0))), \delta(p', \mathbf{vec}_0))) \\
i(t(a, \delta(p, \mathbf{vec}_0))) &\to i(t(a, \pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0)))))
\end{aligned}$$

When a term $i(t(x, y))$ is rewritten, where both x and y are thunks which will instantiate to a and b respectively, this leads to the following inefficiencies:

- i and t are matched 3 times (2 times to discover the thunks, and the last time to find the original match),
- the function symbol t is copied 2 times, because the subterm from the LHS cannot be reused.

This can be repaired by changing the pattern matching code to instantiate the thunks, such that the rules introduced by LS are no longer needed (even though they give a nice model of what is happening). In the pseudo code below, italics indicate the modifications that remove the need for the LS transformation:

```
case x of
i(y): case y of
   t(z1,z2): case z1 of
      a: label1: case z2 of
                    b: continue(a) /* matched ! */
                    thunk: inst(z2); goto label1
                    otherwise: return(x) /* normal form */
       thunk: label2: case z2 of
                 b: inst(z1)
                   case z1 of
                   a) continue(a) /* matched ! */
                   otherwise: return(x) /* normal form */
                 thunk: inst(z2); goto label2
                 otherwise: return(x) /* normal form */
      otherwise: return(x) /* normal form */
   otherwise: ...
otherwise: ...
```

This pattern matching code is bigger than the code for the single rule in the original system, but it is somewhat smaller than the code that corresponds to the transformed system. The increase in time with respect to eager matching is linear in the number of unevaluated thunks encountered during matching, because every unevaluated thunk causes a jump to the state from which the thunk was discovered, and a repeated atomic match operation on the instantiation of the thunk. This means, that laziness is only paid for if it is actually used!

The implementation can be further improved by implementing δ as a tag-bit, $\delta?$ as a bit-test, and **inst** and $\mathbf{vec}_n$ as built-in functions. Finally, the effect of the LR transformation can be achieved by generating slightly different code for right-hand sides.

6.5 Related work

A very early related paper is [Plo75], which gives simulations of call-by-name by call-by-value (eager evaluation), and vice versa, in the context of the λ-calculus. Call-by-name evaluation differs from lazy evaluation (or call-by-need): Thunks are not overwritten with the result of evaluation, but evaluated on every use (which is essential in a language with side-effects).

In the context of functional programs, [Amt93] developed an algorithm to transform call-by-name programs into call-by-value equivalents. In [SW94], dataflow analysis is done in order to minimize thunkification in this context.

In [OLT94], a continuation passing style (cps) transformation of call-by-need into call-by-value equivalents is given. To their knowledge, it is the first. Apart from the fact that they consider a particular λ-calculus, whereas we consider general TRSs, our transformation differs mainly by completely integrating pattern matching of algebraic datatypes in the transformation. It is unclear how much can be gained by taking pattern matching into account in a transformation for a lazy functional implementation. An abstract approach to strictness analysis of algebraic datatypes is investigated in [Ben93].

The effect of our transformations of rewrite systems is somewhat similar in spirit to the use of evaluation transformers in [Bur91]. Not only in theory, but also in practice, our technique does not rely on properties of built-in algebraic datatypes such as lists or trees. In [BM92], some of the techniques in [Bur91] are formulated in the context of continuation passing transformations.

Another approach to obtain better termination properties are the sequential strategies investigated by [O'D77, HL91a]. In this approach, only *needed* redexes are rewritten, i.e., redexes that would be rewritten in any reduction to a normal form. Unfortunately, *neededness* is only well-defined in TRSs that do not have overlapping redexes. This restriction is hard to live with in practice.

To our knowledge, only the Clean [PvE93] and the OBJ3 systems support laziness annotations. Clean supports the annotation of *strict* arguments, OBJ3 [GWM$^+$92] features annotations for the evaluation order of arguments which are somewhat more explicit than ours. It appears that a similar transformation can implement OBJ's annotations.

A rule occuring in the context of an E-unification algorithm, presented in [MMR86], is called "lazy rewriting" in [Klo92]. It might be interesting to investigate if our technique of implementing lazy rewriting on eager machinery is useful in that context.

In CAML (Categorial ML, [CH90]) there are lazy constructors, which can be used to achieve similar effects as our transformation does. However, the transformation of the program must then be carried out manually for the most part (only equivalents of **inst**, δ? and δ are supplied by the implementation).

It is obvious, that our final implementation of lazy term rewriting is similar to the implementation of modern LFPLs As far as we know, these implementations are completely lazy by nature, but are optimized to perform as much eager evaluation as possible.

Therefore, it is appropriate to provide a discussion of the cost of basic datastructures and actions in our scheme, compared with the cost in those implementations. It should be noted that it is *extremely* difficult ([JS89]) to assess the effect of different design choices on performance, so we will only give a qualitative discussion.

- Only a little structure (δ, a thunk constant and a vector containing references to subterms from the left-hand side) occurs below a lazy position in any rhs after the transformation. This is comparable to the frames used in TIM [FW87], or the closures in the STG. Similarly to the latter, our scheme only uses space for the subterms from the LHS that may actually be used later. In the ABC machine [PvE93], complete graphs are built for lazy arguments, which is a drawback compared to all other implementations.

- No runtime cost is incurred when all arguments in the original TRS are annotated eager. Even when all arguments are found to be strict, TIM and STG do a function call to obtain the tag of a constructor term (this is the reason they are called "tagless"), whereas our implementation only needs to dereference a pointer.
- In an implementation that allows overwriting nodes with nodes of arbitrary arity, there is no need for the dreaded indirection nodes ([O'D77, JL92]. In our transformed TRSs, δ fullfills this role; every term (input or rhs) is evaluated exactly once, either by immediate innermost rewriting, or later, by overwriting a δ node. In [JL92], the indirection nodes are also transformed away, but some very complicated analysis is needed to arrive at this result. In the ABC machine, the indirection nodes are indispensable.
- In the rules added by transformation LR, testing if a lazy argument is thunked, is done by rewriting. Even if this is replaced by a bit-test implementation, a subsequent call of **inst** must be done. This is less efficient than the "tagless" reduction which is done in both TIM and STG.
- Unthunking is only done if all eager pattern matching was succesful. Because the order of pattern matching and its effects on evaluation of subterms are fixed in the semantics of LFPLs, this cannot be done in the other implementations.

Taking into account these points, we expect our scheme to perform better than ABC, TIM or the STG, when there is a small number of lazy arguments.

It is clear, that a strictness analyzer can provide laziness annotations (by annotating all arguments that are not found to be strict). However, strictness analyzers being very conservative beasts, this will lead to far too many annotated arguments. So, how much work is involved in finding laziness annotations manually? It is well-known, that even with *lazy* functional programming languages, a thorough understanding of a program is required to make sure that it terminates. In our experience, this level of understanding is adequate to provide complete laziness annotations. Therefore, we hold that programmer-provided laziness annotations are a suitable way of achieving lazy evaluation.

6.6 Conclusions

We have defined *lazy rewriting* and have generalized the notion of *Weak Head Normal Form* to the less operational notion of *Lazy Normal Form.*

We have modeled lazy rewriting by a transformation of term rewriting systems, which avoids rewriting of lazy subterms to a large extent, and completely integrates pattern matching of algebraic datatypes. When all arguments are annotated, the transformed system computes WHNFs.

We derive an efficient implementation on already efficient eager machinery from this model. Our method compares favourably to existing methods.

Our notion of Lazy Normal Forms (LNFs) could also be helpful in an implementation of *abstract rewriting*, as described in [BEØ93], or in the context of *theorem proving.*

Chapter 7

GEL, a Graph Exchange Language

In the previous chapters, we presented techniques to produce executables from Term Rewriting Systems. In order for such executables to cooperate, they should be able to efficiently exchange terms (or directed, acyclic graphs, to be precise). In this chapter, we provide a solution for a generalized problem. Using GEL, the *Graph Exchange Language*, executables can exchange directed (possibly cyclic) graphs, of which the nodes may contain arbitrary data.

7.1 Introduction

Graph-structured data types play a role in a large variety of complex software systems. Especially software performing symbolic manipulation, such as a compiler or a symbolic algebra system, makes use of this kind of data type. Several trends, e.g. the growth of distributed computing and the integration of software developed for complementary purposes, cause a demand for the efficient, language-independent, exchange of graph-structured data.

There are several approaches in existence for the exchange of data, independent of any implementation language, notably ASN.1 (Abstract Syntax Notation One, [CCI]) and the ASCII external representation (ERL) of IDL (Interface Definition Language, [Sno89]).

Of these formalisms, GEL (Graph Exchange Language) bears most similarity in functionality to ERL. For every node in a graph, an ERL description gives a type, a sequence of named edges with corresponding subgraphs, and possibly a label for non-local references.

So why another formalism for the exchange of data? GEL can be characterized by observing how it extends the capabilities of the ERL representation:

- GEL contains a dynamic abbreviation mechanism, which allows the use of very short identifiers in the bulk of the text.
- Instead of labels to identify shared subgraphs or circularities, GEL has relative indices. This enables faster access of shared subgraphs.
- Similarly to the message protocol for trees defined in [DR94], the semantics of GEL

specifies the existence of a stack of subgraphs, leading to an efficient implementation for the important class of DAGs (Directed Acyclic Graphs).

- GEL is compositional: if a graph is composed of several subgraphs, its GEL text can be composed of the GEL texts of its subgraphs. In ERL, the labels in the texts of the subgraphs have to be made unique before composing.

We claim that in many cases, GEL removes the need to implement a more efficient exchange protocol for production versions of a system ([Sno89], page 141). Thus, components in the prototype phase can be mixed freely with production versions of other components.

GEL is more austere than ERL, because there are no built-in datatypes `string` or `integer`. At reasonable costs, these datatypes can be implemented on top of GEL. GEL deals exclusively with the compressed exchange of graph-structured data.

We will first give an informal overview of GEL in Section 7.2. In Sections 7.4 to 7.8, we will present the full GEL language by way of a guided tour through an algebraic specification of the GEL reader in the ASF+SDF formalism, which is briefly introduced in Section 7.3.

In Section 7.9, we present an algorithm for writing GEL texts. In Section 7.11, we go into the design decisions that led to the ultimate design of GEL, and in Section 7.12, we present timings of our experimental C implementation.

In Appendices B and B.1, we give a binary encoding of GEL, and the interface description of our implementation as a library for use with the `C` programming language [KR78].

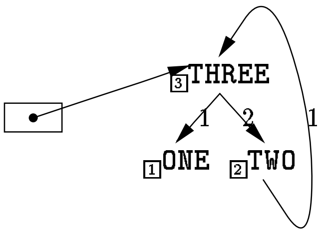
7.2 An informal overview of GEL

GEL describes rooted, directed, connected graphs with typed (labeled) nodes and ordered edges, called *term graphs* in [BvEJ+87]. All graph-structured datatypes can easily be mapped to GEL graphs.

As an informal introduction, we have drawn a suggestive picture of a GEL graph in Figure 7.1. The rectangular box with the arrow is a pointer to the root of the graph. The types of the nodes in this graph are `ONE`, `TWO` and `THREE`. For easy reference, the nodes are numbered in their lower left corner. The direction of an edge is indicated by its arrowhead, and the ordering of the edges is indicated by numbers (1,2).

Basically, a GEL text defines abbreviations for the compression of types, and gives directives to build nodes with edges pointing to other nodes. GEL is a stack-based language, i.e., the directive to build a node assumes that the nodes referred to are on top of a stack in the appropriate order. This allows for a very short encoding (postfix) in case the graphs are trees. If the graphs fall in a more general class, special elements are pushed onto the stack, taking care of multiple or cyclic references.

As a short example, consider the GEL text in Figure 7.1. The text in this figure is a readable representation of the actual binary encoding of GEL defined in Appendix B. The first line in this figure defines an abbreviation `a` for the type `ONE`, which has no edges. The



```
! a : 0 = ONE <RET>
a <RET>
! b : 1 = TWO <RET>
> 1 <RET>
b <RET>
! c : 2 = THREE <RET>
c <RET>
```

Figure 7.1: A rooted graph and its GEL encoding

second line builds a node of this type, and puts it on the stack. The third line defines an abbreviation `b` for the type `TWO`, which has one edge. The fourth line puts a future reference on the stack, which is used in the fifth line to build a node of the type `TWO`, with one edge. The fifth line defines an abbreviation for the type `THREE`, which has two edges. This abbreviation is used in the sixth line to build a node with two edges. The last reference remaining on the stack is interpreted as the root of the graph, so the GEL text in the right part of Figure 7.1 describes the graph in the left part of Figure 7.1.

7.3 A quick overview of ASF+SDF

ASF+SDF is a specification formalism for describing all syntactic and semantic aspects of (formal) languages. It is an amalgamation of the formalisms SDF [HHKR89] for describing syntax, and ASF [BHK89b] for describing semantics.

ASF is a conventional algebraic specification formalism providing notions like first-order signatures, import/export, variables, and conditional equations. The meaning of ASF specifications is based on their initial algebra semantics. If specifications satisfy certain criteria, they can be executed as a term rewriting system.

SDF introduces the idea of a “syntactic front-end” for terms and equations defined over a first-order signature. This creates the possibility to write first-order terms as well as equations in arbitrary concrete syntactic forms: from a given SDF definition for some context-free grammar, a fixed mapping from strings to terms can be derived.

An ASF+SDF specification consists of a sequence of named modules. Each module may contain:

imports of other modules;

sort declarations defining the sorts of a signature;

lexical syntax defining layout conventions and lexical tokens;

context-free syntax defining the concrete syntactic forms of the functions in the signature;

variables to be used in equations. In general, each variable declaration has the form of a regular expression and defines the class of all variables whose name is described by the regular expression;

equations defining the meaning of the functions declared in the context-free syntax section.

An unusual feature, associative *lists*, will be used in this paper, and deserves some further explanation. In the description of context-free grammars one frequently encounters the notion of iteration or list, in order to describe syntactic constructs like statement-list, parameter-list, declaration-list, etc. In ASF+SDF this notion is provided at the syntactic as well as at the semantic level. At the syntactic level, one can define, for instance, a "list of zero or more identifiers separated by comma's". At the semantic level, variables over such lists may be declared and used in equations. Semantically, lists can always be eliminated. Operationally, the matching of a list structure is achieved by local backtracking during term rewriting [Hen89].

For the presentation of our specification, we use Eelco Vissers "ToLaTeX" package. Apart from providing a nice layout for terms and equations, it prints section numbers in the upper right corner of imports, for easy reference.

7.4 An overview of the specification of GEL

In Figure 7.2, the structure of the GEL specification is displayed. Starting at the bottom, and proceeding from left to right, we have the following modules.

Ints Integers are used to number the nodes and edges in a graph. The specification of this module can be found in [Wal94].

Layout Layout is needed in order to present the equations of modules in a readable way. However, the GEL formalism is very restrictive with respect to layout. Therefore, much care is taken to only import Layout where necessary. Because this module is not specific to GEL, it is given in Appendix B.3.

Gel-types specifies basic notions which occur both in graphs and in their GEL descriptions. Section 7.5.1 explains the module.

Graphs In this module the subject matter of GEL is defined formally. A discussion can be found in Section 7.5.2.

Gel Here, the syntax of commands occurring in GEL is defined. See Section 7.7.1 for an overview of the commands. The full lexical definition of sorts is deferred to Full-Gel-syntax (see below).

Stacks During the construction of graphs, stacks of subgraphs are used. These stacks and the operations on them are defined in this module. Section 7.6.2 gives more detail.

Tables While reading a GEL text, the GEL reader must maintain a correspondence between abbreviated and full type names. The datatype needed for this is specified in the module Tables. Section 7.6.1 contains the full specification.

Full-Gel-syntax The full definition of the readable form of GEL. Ideally, this definition should be given by the module Gel, but this would cause parsing problems in the equations of Gel-read. In Section B.2, these subtleties are explained.

Gel-read Finally, the effect of the commands available in GEL is defined by a specification of the transitions of an abstract machine. Given the auxiliary functions defined in the preceding modules, every command can be specified in a single equation. Section 7.7.2 elaborates on this module.

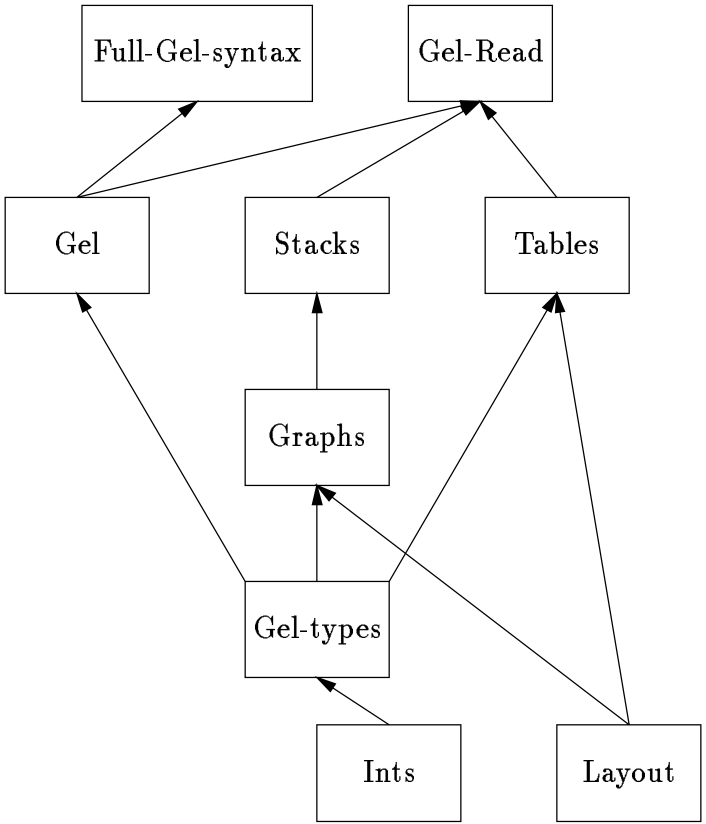


Figure 7.2 The structure of the GEL specification

7.5 Basics: Gel-types and Graphs

We start with the specification of the basics. Gel-types are used to specify the contents of the nodes occurring in graphs.

7.5.1 Gel-types

```
Module Gel-types
imports  Ints(B.3)
exports
  sorts TYPE SHORT-TYPE RET SPACE
  lexical syntax
    [ '0-9a-zA-Z]+            → TYPE
    [ 'a-zA-Z][a-zA-Z0-9]*   → SHORT-TYPE
  context-free syntax
    "*" → INT
  variables
    Type [0-9  ']*→ TYPE
    Space         → SPACE
    Ret           → RET
hiddens
  lexical syntax
    [␣\t\n]             → LAYOUT
    "//"~[\n]*[\n]      → LAYOUT
    "%%"~[\n]*[\n]      → LAYOUT
equations


    * = − 1                                                    [varyad]
```

As far as GEL is concerned, the possible types of nodes are an uninterpreted parameter. In the main part of our specification, we model this by a lexical definition of the sort TYPE that admits identifiers with quotes. Only in the module Full-Gel-syntax, this is extended to any string that does not contain a newline. In the binary implementation of GEL (see Appendix B), a TYPE can be any (length-encoded) sequence of bytes.

In GEL, abbreviations can be introduced for types, where the lexical syntax for abbreviations is given by the sort SHORT-TYPE. In the binary implementation of GEL, a SHORT-TYPE is replaced by a binary encoding of a number (index in a table).

In order to parse the equations without exporting lexical syntax for LAYOUT, LAYOUT is defined locally.

In order to stress the non-numeric meaning of -1 (see Section 7.5.2) we introduce an alias $*$.

7.5.2 Graphs

Module Graphs
imports Ints$^{(B.3)}$ Gel-types$^{(7.5.1)}$ Layout$^{(B.3)}$
exports
sorts EDGE EDGES NODE NODES GRAPH INTS
context-free syntax
node INT *of type* TYPE *with sig* INT *and edges* EDGES → NODE
“[” {NODE “,”}* “]” → NODES

root is INT *in* NODES → GRAPH
error_graph → GRAPH

INT “→” INT → EDGE
“{” {EDGE “,”}* “}” → EDGES
variables
Edge [*0-9* ′]* → EDGE
Edge [*0-9* ′]*“*” → {EDGE “,”}*
Graph [*0-9* ′]* → GRAPH
Node [*0-9* ′]* → NODE
Node [*0-9* ′]*“*” → {NODE “,”}*
hiddens
context-free syntax
occurs INT *in* NODES → INT
occurs INT *in* EDGES → INT
“{” {INT “,”}* “}” → INTS
occurs INT *in* INTS → INT
NODES *close with* INTS → INTS
num-edges EDGES → INT
variables
Int [*0-9* ′]*“*” → {INT “,”}*

In the specification of GEL's graphs, the nodes are taken as the basis of description. A node has a number (INT) for identification purposes, a type (TYPE) which is uninterpreted in our specification, an arity indication (INT) and a number (possibly zero) of edges (EDGE).

The arity indication is redundant for the specification of the graph structure, but it enables a more efficient GEL representation (see Section 7.11). Non-negative arities specify the number of edges, an arity of `-1` signifies a *varyadic* arity, meaning that the node can have any number of edges.

An edge is a pair of integers (INT), where the first integer indicates the node of departure, and the second integer indicates the destination node.

A GEL graph consists of a number of nodes, with a (positive, see below) integer indicating the root.

We have left unspecified, that a consistent renumbering of the nodes and the root (cf. alpha-conversion in the λ-calculus) yields exactly the same graph structure.
Given these definitions, we present in Figure 7.3 a term representation of the GEL graph in Figure 7.1.

$$\begin{array}{l}
\mathit{root\ is}\ 3\\
\mathit{in}\ [\mathit{node}\ 1\ \mathit{of\ type}\ \mathtt{ONE}\ \mathit{with\ sig}\ 0\ \mathit{and\ edges}\ \{\},\\
\quad \mathit{node}\ 2\ \mathit{of\ type}\ \mathtt{TWO}\ \mathit{with\ sig}\ 1\ \mathit{and\ edges}\ \{1 \longrightarrow 3\},\\
\quad \mathit{node}\ 3\ \mathit{of\ type}\ \mathtt{THREE}\\
\quad \mathit{with\ sig}\ 2\\
\quad \mathit{and\ edges}\ \{1 \longrightarrow 2,\\
\qquad\qquad\quad 2 \longrightarrow 3\}]
\end{array}$$

Figure 7.3 Term representation of the graph in Figure 7.1

equations

One element of the sort GRAPH is used as an error element:

$$\mathit{error\_graph} = \mathit{root\ is}\ 0\ \mathit{in}\ [] \qquad \text{[error-graph-0]}$$

Node numbers are positive integers

$$\frac{X_4 < 1 = 1}{\begin{array}{l}\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^{*\prime},\\ \qquad \mathit{node}\ X_4\ \mathit{of\ type\ Type\ with\ sig}\ X_2\\ \qquad \mathit{and\ edges}\ \{\mathit{Edge}^{*}\},\\ \qquad \mathit{Node}^{*\prime\prime}] \qquad\qquad\qquad = \mathit{error\_graph}\end{array}} \qquad \text{[error-1]}$$

The root of the graph should point into the graph

$$\frac{\mathit{occurs}\ X\ \mathit{in}\ [\mathit{Node}^{*}] = 0}{\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^{*}] = \mathit{error\_graph}} \qquad \text{[error-2]}$$

All edges should point into the graph

$$\frac{\begin{array}{l}\qquad [\mathit{Node}^{*}] = [\mathit{Node}^{*\prime},\\ \qquad\qquad \mathit{node}\ X_4\ \mathit{of\ type\ Type\ with\ sig}\ X_2\\ \qquad\qquad \mathit{and\ edges}\ \{\mathit{Edge}^{*},\, X_3 \longrightarrow X_1,\, \mathit{Edge}^{*\prime}\},\\ \qquad\qquad \mathit{Node}^{*\prime\prime}],\\ \mathit{occurs}\ X_1\ \mathit{in}\ [\mathit{Node}^{*}] = 0\end{array}}{\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^{*}] = \mathit{error\_graph}} \qquad \text{[error-3]}$$

The number of edges should correspond to the signature

$$\frac{\begin{array}{l}\qquad [\mathit{Node}^{*}] = [\mathit{Node}^{*\prime},\\ \qquad\qquad \mathit{node}\ X_4\ \mathit{of\ type\ Type\ with\ sig}\ X_2\\ \qquad\qquad \mathit{and\ edges}\ \{\mathit{Edge}^{*}\},\\ \qquad\qquad \mathit{Node}^{*\prime\prime}],\\ \qquad X_2 < 0 = 0,\\ \mathit{num\text{-}edges}\ \{\mathit{Edge}^{*}\} \neq X_2\end{array}}{\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^{*}] = \mathit{error\_graph}} \qquad \text{[error-4]}$$

Every node should have a unique id

$$\begin{array}{l} \mathit{root\ is}\ X_1\ \mathit{in}\ [\mathit{Node}^*, \\ \qquad \mathit{node}\ X\ \mathit{of\ type\ Type\ with\ sig}\ X_2\ \mathit{and\ edges}\ \{\mathit{Edge}^*\}, \\ \qquad \mathit{Node}^{*\prime}, \\ \qquad \mathit{node}\ X\ \mathit{of\ type\ Type}'\ \mathit{with\ sig}\ {X_2}'\ \mathit{and\ edges}\ \{\mathit{Edge}^{*\prime}\}, \\ \qquad \mathit{Node}^{*\prime\prime}] \qquad = \mathit{error\_graph} \end{array} \qquad \text{[error-5]}$$

Every node should be reachable from the root

$$\frac{\begin{array}{l} [\mathit{Node}^*]\ \mathit{close\ with}\ \{X\} = \{\mathit{Int}^*\}, \\ \qquad [\mathit{Node}^*] = [\mathit{Node}^{*\prime}, \\ \qquad\qquad \mathit{node}\ X_1\ \mathit{of\ type\ Type\ with\ sig}\ X_2 \\ \qquad\qquad \mathit{and\ edges}\ \{\mathit{Edge}^*\}, \\ \qquad\qquad \mathit{Node}^{*\prime\prime}], \\ \quad \mathit{occurs}\ X_1\ \mathit{in}\ \{\mathit{Int}^*\} = 0 \end{array}}{\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*] = \mathit{error\_graph}} \qquad \text{[error-6]}$$

Labels start from 1

$$\frac{X_1 < 1 = 1}{\begin{array}{l} \mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*, \\ \qquad \mathit{node}\ X_1\ \mathit{of\ type\ Type\ with\ sig}\ X_2 \\ \qquad \mathit{and\ edges}\ \{\mathit{Edge}^*, X_1 \rightarrow {X_1}', \mathit{Edge}^{*\prime}\}, \\ \qquad \mathit{Node}^{*\prime}] \qquad = \\ \mathit{error\_graph} \end{array}} \qquad \text{[error-7]}$$

Every label should occur only once on a node

$$\begin{array}{l} \mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*, \\ \qquad \mathit{node}\ X_1\ \mathit{of\ type\ Type\ with\ sig}\ X_2 \\ \qquad \mathit{and\ edges}\ \{\mathit{Edge}^*, X_1 \rightarrow {X_1}', \mathit{Edge}^{*\prime}, X_1 \rightarrow {X_1}'', \mathit{Edge}^{*\prime\prime}\}, \\ \qquad \mathit{Node}^{*\prime}] \qquad = \\ \mathit{error\_graph} \end{array} \qquad \text{[error-8]}$$

Labels should be contiguous

$$\frac{\begin{array}{r} X_1 < 2 = 0, \\ \mathit{occurs}\ X_1 - 1\ \mathit{in}\ \{\mathit{Edge}^*, \mathit{Edge}^{*\prime}\} = 0 \end{array}}{\begin{array}{l} \mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*, \\ \qquad \mathit{node}\ X_1\ \mathit{of\ type\ Type\ with\ sig}\ X_2 \\ \qquad \mathit{and\ edges}\ \{\mathit{Edge}^*, X_1 \rightarrow {X_1}', \mathit{Edge}^{*\prime}\}, \\ \qquad \mathit{Node}^{*\prime}] \qquad = \\ \mathit{error\_graph} \end{array}} \qquad \text{[error-9]}$$

Auxiliary functions define occurence in sets and closure under traversal

$$\frac{\mathit{occurs}\ X_3\ \mathit{in}\ \{\mathit{Int}^*, X, \mathit{Int}^{*\prime}\} = 0}{\begin{array}{l}[\mathit{Node}^*,\\ \quad \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\\ \quad \mathit{and\ edges}\ \{\mathit{Edge}^*, X_2 \rightarrow X_3, \mathit{Edge}^{*\prime}\},\\ \quad \mathit{Node}^{*\prime}]\ \mathit{close\ with}\ \{\mathit{Int}^*, X, \mathit{Int}^{*\prime}\} \qquad =\\ [\mathit{Node}^*,\\ \quad \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\\ \quad \mathit{and\ edges}\ \{\mathit{Edge}^*, X_2 \rightarrow X_3, \mathit{Edge}^{*\prime}\},\\ \quad \mathit{Node}^{*\prime}]\ \mathit{close\ with}\ \{\mathit{Int}^*, X, X_3, \mathit{Int}^{*\prime}\}\end{array}}$$ [closure]

$[\mathit{Node}^*]\ \mathit{close\ with}\ \{\mathit{Int}^*\} = \{\mathit{Int}^*\}$ **otherwise** [closure]

$\mathit{occurs}\ X\ \mathit{in}\ \{\mathit{Int}^*, X, \mathit{Int}^{*\prime}\} = 1$ [occurs-ints-0]

$\mathit{occurs}\ X\ \mathit{in}\ \{\mathit{Int}^*\} = 0$ **otherwise** [occurs-ints]

$\mathit{occurs}\ X\ \mathit{in}\ [\mathit{Node}^*, \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\ \mathit{and\ edges}\ \{\mathit{Edge}^*\}, \mathit{Node}^{*\prime}] =$ [occurs-0]
1

$\mathit{occurs}\ X\ \mathit{in}\ [\mathit{Node}^*, \mathit{node}\ X_1\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_2\ \mathit{and\ edges}\ \{\mathit{Edge}^*\}, \mathit{Node}^{*\prime}] =$ [occurs]
0 **otherwise**

$\mathit{num\text{-}edges}\ \{\} = 0$ [num-edges-0]

$\mathit{num\text{-}edges}\ \{\mathit{Edge}, \mathit{Edge}^*\} = 1 + \mathit{num\text{-}edges}\ \{\mathit{Edge}^*\}$ [num-edges-1]

7.6 The machine components

In Sections 7.6.1 and 7.6.2, we will specify the components of the GEL abstract machine. Here, we first give an informal overview. As a visual aid, a state of the GEL reading machine is depicted in Figure 7.4. A state consists of an abbreviation table, a stack of graph references, and an (unfinished) graph.

At the top of the figure, the abbreviation table is shown, associating an arity and a type with every abbreviation. For example, the abbreviation **a** is associated with a signature indicating an arity of 2, and a type `APPLY`.

Below the abbreviation table, on the left, the stack of graph references is shown, and on the right, we find the graph under construction. The nodes in the graph are shown as a type, tagged with a unique node number (only for reference, graphs are equivalent under node renumberings). Formally, the arity should also be indicated, but in our examples it follows from the context. The edges in the graph are arrows, tagged with their index in the edge-ordering.

SHORT-TYPE	SIG	TYPE
a	2	`APPLY`
b	0	`ONE`
c	-1	`LIST`
d	2	`PLUS`

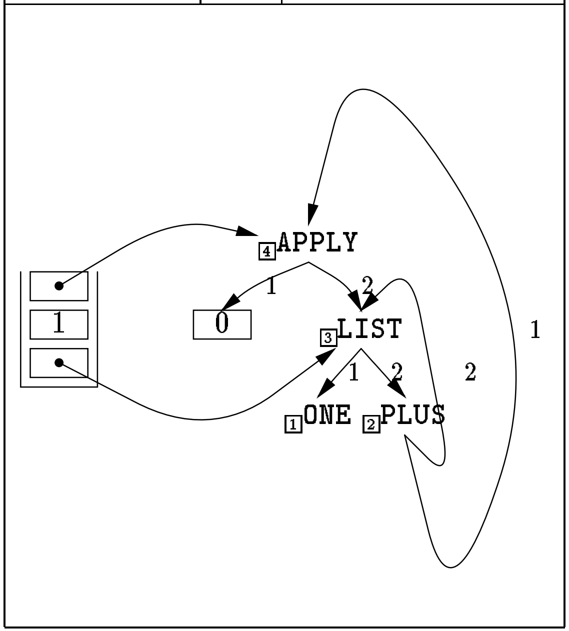


Figure 7.4 A GEL graph under construction

The stack contains ordinary references and ‘future’ references. The ordinary references are depicted as rectangles from which arrows emanate, in this example to the subgraphs rooted at `APPLY` and `LIST`. The ‘future’ references are depicted as rectangles containing numbers, indicating how far in the future the graph will be built. There is a future reference with number "1" on the stack, and a future reference with number "0" in the graph. The number in a future reference specifies the number of build operations to be performed before the indicated graph is constructed.

7.6.1 Tables

Module Tables
imports Layout$^{(B.3)}$ Gel-types$^{(7.5.1)}$
exports
  sorts TABLE ITEM
  context-free syntax
    sig of SHORT-TYPE *in* TABLE → INT
    type of SHORT-TYPE *in* TABLE → TYPE

define SHORT-TYPE *of type* TYPE *with sig* INT *in* TABLE → TABLE
empty-table → TABLE
variables
Table $[0\text{-}9\ ']*$ → TABLE
Short-Type $[0\text{-}9\ ']*$→ SHORT-TYPE

The concrete functions for the table should be hidden, but then the example terms further in the text become unparseable.

context-free syntax
abbr SHORT-TYPE *of type* TYPE *with sig* INT → ITEM
"[" {ITEM ","}* "]" → TABLE
variables
Item $[0\text{-}9\ ']*$ → ITEM
Item $[0\text{-}9\ ']*$"*"→ {ITEM ","}*

equations

$$\mathit{empty\text{-}table} = [\,] \qquad \text{[empty-table-0]}$$

$$\mathit{define\ Short\text{-}Type\ of\ type\ Type\ with\ sig\ X\ in}\ [\mathit{Item}^*] = [\mathit{abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig}\ X, \mathit{Item}^*] \qquad \text{[define-0]}$$

$$\mathit{sig\ of\ Short\text{-}Type\ in}\ [\mathit{abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig}\ X, \mathit{Item}^*] = X \qquad \text{[sig-0]}$$

$$\frac{\mathit{Short\text{-}Type} \neq \mathit{Short\text{-}Type}'}{\mathit{sig\ of\ Short\text{-}Type\ in}\ [\mathit{abbr\ Short\text{-}Type}'\ \mathit{of\ type\ Type\ with\ sig}\ X, \mathit{Item}^*] = \mathit{sig\ of\ Short\text{-}Type\ in}\ [\mathit{Item}^*]} \qquad \text{[sig-1]}$$

$$\mathit{sig\ of\ Short\text{-}Type\ in\ Table} = -\,2 \qquad \textbf{otherwise} \qquad \text{[sig-error]}$$

$$\mathit{type\ of\ Short\text{-}Type\ in}\ [\mathit{abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig}\ X, \mathit{Item}^*] = \mathit{Type} \qquad \text{[type-0]}$$

$$\frac{\mathit{Short\text{-}Type} \neq \mathit{Short\text{-}Type}'}{\mathit{type\ of\ Short\text{-}Type\ in}\ [\mathit{abbr\ Short\text{-}Type}'\ \mathit{of\ type\ Type\ with\ sig}\ X, \mathit{Item}^*] = \mathit{type\ of\ Short\text{-}Type\ in}\ [\mathit{Item}^*]} \qquad \text{[type-1]}$$

$$\mathit{type\ of\ Short\text{-}Type\ in}\ [\,] = \texttt{error} \qquad \text{[type-2-error]}$$

A new abbreviation can be defined by *define*. It is possible that a certain SHORT-TYPE is defined several times. However, the equations for the access functions specify that only the last definition matters.

Given a SHORT-TYPE, the associated TYPE and SIG can be found by applying the functions *type of* and *sig of*.

7.6.2 Stacks

Module Stacks
imports Graphs$^{(7.5.2)}$
exports
sorts STACK
context-free syntax
empty → STACK
push(INT, STACK) → STACK

top(STACK) → INT
top(INT, STACK) → INT
pop(STACK) → STACK
pop(INT, STACK) → STACK

drop(INT, STACK) → STACK
take(INT, STACK) → EDGES
variables
Stack [*0-9* ′]∗→ STACK
equations

$top(push(X, Stack)) = X$ [top-0]

$top(empty) = 0$ [top-error]

$top(0, Stack) = top(Stack)$ [top-1]

$$\frac{X < 1 = 0}{top(X, Stack) = top(X - 1, pop(Stack))}$$ [top-n]

$$\frac{X < 0 = 1}{top(X, Stack) = 0}$$ [top-error]

$pop(push(X, Stack)) = Stack$ [pop-0]

$pop(empty) = empty$ [pop-error]

$$\frac{X < 1 = 0}{pop(X, Stack) = pop(X - 1, pop(Stack))}$$ [pop-n]

$$\frac{X < 1 = 1}{pop(X, Stack) = Stack}$$ [pop-1-error]

$drop(X, Stack) = push(top(Stack), pop(X + 1, Stack))$ [drop-0]

$$\frac{\begin{array}{c} X < 1 = 0, \\ take(X - 1, pop(Stack)) = \{Edge^*\} \end{array}}{take(X, Stack) = \{Edge^*, X \rightarrow top(Stack)\}} \quad \text{[take-1]}$$

$$take(X, Stack) = \{\} \qquad \textbf{otherwise} \quad \text{[take]}$$

Apart from the operations *drop* and *take*, and the use of the index 0 to reference the top, this stack specification is completely standard.

Stacks are built from the constructor functions *empty* and *push*. The stack is either empty, or it contains references (by number) to nodes in the graph. If a number on the stack is larger than the number of any node in the graph, it is taken to be a forward reference.

The functions *top* and *pop* are defined in a fairly usual way. However, the functions *drop* and *take* are unusual. The function *drop* pops elements off the stack, but preserves the topmost element. The function *take* produces a list of edges from a stack and a SIG.

7.7 GEL syntax and semantics

Now we will discuss GEL command by command. We will give the semantics of GEL by specifying how the state of the abstract machine defined in Section 7.6 is affected by reading a single line. Given the specification of the abstract machine, we need only one equation for the specification of the semantics of one GEL command.

7.7.1 An overview of GEL commands

We will present the overview of GEL commands by annotating separate context-free syntax sections in the module Gel.

Module Gel
imports Gel-types$^{(7.5.1)}$
exports
  sorts GEL-ITEM GEL
  context-free syntax
    GEL-ITEM∗ → GEL

A full GEL text is a sequence of GEL-ITEMs. Every GEL-ITEM is terminated by a carriage return. Because of lexical syntax problems (see Appendix B.2) this return is put in a sort RET, which is fully defined only in module Full-Gel-syntax. In all other modules, only variables and meta-variables of this sort can be used.

We now proceed with a description of the individual GEL commands.

Comments

context-free syntax

"%" TYPE RET → GEL-ITEM

Comments are ignored. The library in Appendix B.1 inserts a comment with version information.

The abbreviation command

context-free syntax

"!" SHORT-TYPE ":" INT "=" TYPE RET → GEL-ITEM

Primarily, the compactness of GEL is achieved by its abbreviation mechanism. For every combination of type and arity occurring in a graph described by a GEL text, an abbreviation is introduced by the abbreviation command. The syntax for the abbreviation command is introduced in the context-free syntax section above. The effect of this command is defined in Section 7.7.3.

To GEL, the type of a node is uninterpreted, but for the graph structure, it *is* important how many edges depart from a node. Somewhat redundantly, this is specified both by an arity (sort INT) in the definition of an abbreviation and by the actual edges departing from the node. Positive values denote fixed arities, -1 denotes varyadic arity (with an alias *), and all other values are used as error values. In Section 7.11, we will discuss the concerns leading to this redundant specification.

The build commands

context-free syntax

SHORT-TYPE RET → GEL-ITEM
SHORT-TYPE SPACE INT RET → GEL-ITEM

There are two versions of the build command, one for fixed and one for varyadic arities. The varyadic variant has an integer argument specifying the actual number of edges to make. The effect of these commands is defined in Section 7.7.4.

The copy commands

context-free syntax

"#" INT RET → GEL-ITEM
">" INT RET → GEL-ITEM

For the expression of sharing and circularities, there are two copy commands, one for backward references (introduced by '#') and one for forward references (introduced by '>'). The parameter of a backward reference denotes an offset in the stack, where the reference to be copied can be found. The parameter of a forward reference denotes the number of build commands to be processed until the actual node will be produced.

The drop command

context-free syntax

"∗" INT RET → GEL-ITEM

When nodes with non-zero arities are built, references are popped off the stack. It is not always possible to put the references on the stack in such a way that only one reference remains on the stack after the last build command. To this purpose, the drop command removes a number of references just below the top reference on the stack.

7.7.2 GEL semantics

Module Gel-read
imports Gel$^{(7.7.1)}$ Graphs$^{(7.5.2)}$ Stacks$^{(7.6.2)}$ Tables$^{(7.6.1)}$
exports
context-free syntax
read GEL → GRAPH
hiddens
context-free syntax
read GEL *with next* INT "," *stack* STACK "," *abbreviations* TABLE *and nodes* NODES → GRAPH
variables
Gel-Item [*0-9* ′]∗"∗" → GEL-ITEM∗
Gel [*0-9* ′]∗ → GEL
equations

read Gel = *read Gel with next* 1, *stack empty*, *abbreviations empty-table and nodes* [] [read-0]

read with next X, *stack push*(X', *empty*), *abbreviations Table and nodes* [*Node**] =
root is X' *in* [*Node**] [extract-0]

read Gel with next X, *stack Stack*, *abbreviations Table and nodes* [*Node**] =
error_graph **otherwise** [extract]

As a hidden function, the module Gel-read contains the constructor function for a state of the GEL machine. The first argument is the GEL-text still to be read, the second argument is the number of the next node to build, the third argument a stack of references to subgraphs, the fourth argument a table of abbreviations for types and signatures, and the fifth arguments contains the nodes that have been built until now. Final states have exactly one reference on the stack, and an empty GEL text, otherwise the input GEL text was erroneous. In the following subsections, we will use a running example to illustrate the effects of the commands. The initial state of the GEL machine reading this example is given in Figure 7.5.

7.7.3 Effects of the abbreviation command

Abbreviations are handled by the equation

read ! *Short-Type* : X_1 = *Type Ret*
    Gel-Item*
with next X, *stack Stack*, *abbreviations Table and nodes* [*Node**] = [read-abbr]
read Gel-Item* *with next* X, *stack Stack*,
    abbreviations define Short-Type of type Type with sig X_1 *in Table*
and nodes [*Node**]

In this equation, only the abbreviation table is updated. Thus, after reading abbreviations for a type `ONE` with arity 0, a type `LIST` of varyadic arity, and a type `PLUS` with named edges `left` and `right`, the state of the GEL machine is as shown in Figure 7.6.

read ! a : 0 = ONE <RET>
    ! b : $*$ = LIST <RET>
    ! c : 2 = PLUS <RET>
    a <RET>
    b <SPACE> 0 <RET>
    c <RET>
    # 0 <RET>
    > 1 <RET>
    c <RET>
    # 1 <RET>
    b <SPACE> 2 <RET>
    $*$ 1 <RET>
with next 1,
    stack empty,
        abbreviations []
and nodes []

Figure 7.5 An initial state of the GEL reader

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

```
read a <RET>
     b <SPACE> 0 <RET>
     c <RET>
     # 0 <RET>
     > 1 <RET>
     c <RET>
     # 1 <RET>
     b <SPACE> 2 <RET>
     * 1 <RET>
with next 1,
          stack empty,
                abbreviations [abbr c of type PLUS with sig 2,
                               abbr b of type LIST with sig *,
                               abbr a of type ONE with sig 0]
and nodes []
```

Figure 7.6: The GEL machine after reading 3 abbreviations

7.7.4 Effects of the build commands

Build commands are handled by the equations

```
                X = sig of Short-Type in Table,
            X < 0 = 0
------------------------------------------------------------------   [read-bld-fix]
read Short-Type Ret
     Gel-Item*
with next X1, stack Stack, abbreviations Table and nodes [Node*] =
read Gel-Item*
with next X1 + 1,
          stack push(X1, pop(X, Stack)),
                abbreviations Table
and nodes [Node*, node X1 of type type of Short-Type in Table
                  with sig X
                  and edges take(X, Stack)]
```

```
          sig of Short-Type in Table < 0 = 1
------------------------------------------------------------------   [read-bld-fix-error]
read Short-Type Ret
     Gel-Item*
with next X1, stack Stack, abbreviations Table and nodes [Node*] =
error_graph
```

$$\frac{\textit{sig of Short-Type in Table} = -1}{\begin{array}{l} \textit{read Short-Type Space X Ret} \\ \quad \textit{Gel-Item}^* \\ \textit{with next } X_1\textit{, stack Stack, abbreviations Table and nodes } [\textit{Node}^*] = \\ \textit{read Gel-Item}^* \\ \textit{with next } X_1 + 1, \\ \qquad \textit{stack push}(X_1, \textit{pop}(X, \textit{Stack})), \\ \qquad\quad \textit{abbreviations Table} \\ \textit{and nodes } [\textit{Node}^*, \textit{node } X_1 \textit{ of type type of Short-Type in Table} \\ \qquad\qquad \textit{with sig sig of Short-Type in Table} \\ \qquad\qquad \textit{and edges take}(X, \textit{Stack})] \end{array}}$$

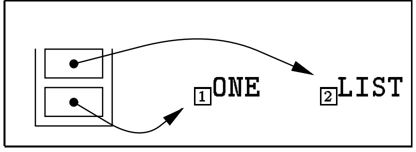
[read-bld-var]

$$\frac{\textit{sig of Short-Type in Table} \neq -1}{\begin{array}{l} \textit{read Short-Type Space X Ret} \\ \quad \textit{Gel-Item}^* \\ \textit{with next } X_1\textit{, stack Stack, abbreviations Table and nodes } [\textit{Node}^*] = \\ \textit{error\_graph} \end{array}}$$

[read-bld-var-error]

The equations describing the effect on the GEL machine are very similar. For types of fixed arity (equation **read-bld-fix**), the number of items to be popped of the stack is looked up in the abbreviation table, whereas for varyadic types (equation **read-bld-var**), this number is taken from the command. Similarly, the edge names are taken from the signature for types of fixed arity, whereas for varyadic types the edge names are numbered according to the command. Because the build command actually builds a new node, the count of nodes is increased by one in both equations. In Figure 7.7, the effect of reading of two build commands containing abbreviations of a fixed and a varyadic type is shown.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

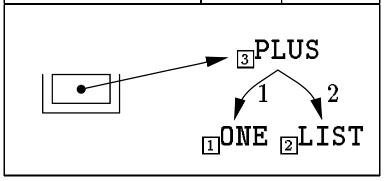


```
read c <RET>
    # 0 <RET>
    > 1 <RET>
    c <RET>
    # 1 <RET>
    b <SPACE> 2 <RET>
    * 1 <RET>
with next 3,
        stack push(2, push(1, empty)),
            abbreviations [abbr c of type PLUS with sig 2,
                           abbr b of type LIST with sig *,
                           abbr a of type ONE with sig 0]
and nodes [node 1 of type ONE with sig 0 and edges {},
        node 2 of type LIST with sig * and edges {}]
```

Figure 7.7 The GEL machine after 3 abbreviations and 2 builds

Note that after each build command, a reference to the node built is left on the stack. These references are popped on build commands for non-zero arities, e.g. reading of a `c` on the next line of our example results in the state displayed in Figure 7.8.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS



read $\#$ 0 <RET>
$>$ 1 <RET>
`c` <RET>
$\#$ 1 <RET>
`b` <SPACE> 2 <RET>
$*$ 1 <RET> *with next* 4,
stack push(3, *empty*),
abbreviations [*abbr* `c` *of type* `PLUS` *with sig* 2,
abbr `b` *of type* `LIST` *with sig* $*$,
abbr `a` *of type* `ONE` *with sig* 0]
and nodes [*node* 1 *of type* `ONE` *with sig* 0 *and edges* {},
node 2 *of type* `LIST` *with sig* $*$ *and edges* {},
node 3 *of type* `PLUS` *with sig* 2
and edges {1 → 1,
2 → 2}]

Figure 7.8 The GEL machine after 3 abbreviations and 3 builds

7.7.5 Effects of the copy commands

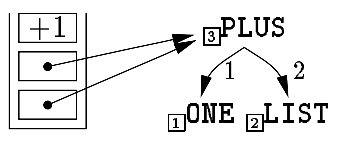
The meaning of the copy commands is specified in the equations

$$\begin{array}{l} \textit{read} \; \# \; X \; \textit{Ret} \\ \quad \textit{Gel-Item}^* \\ \textit{with next } X_1, \textit{ stack Stack}, \textit{ abbreviations Table and nodes } [\textit{Node}^*] = \qquad \text{[read-scopy]} \\ \textit{read Gel-Item}^* \textit{ with next } X_1, \textit{ stack push}(top(X, \textit{Stack}), \textit{Stack}), \textit{ abbreviations Table and nodes } [\textit{Node}^*] \end{array}$$

$$\begin{array}{l} \textit{read} > X \; \textit{Ret} \\ \quad \textit{Gel-Item}^* \\ \textit{with next } X_1, \textit{ stack Stack}, \textit{ abbreviations Table and nodes } [\textit{Node}^*] = \qquad \text{[read-fcopy]} \\ \textit{read Gel-Item}^* \textit{ with next } X_1, \textit{ stack push}(X_1 + X, \textit{Stack}), \textit{ abbreviations Table and nodes } [\textit{Node}^*] \end{array}$$

In ASF+SDF terms, the subsequent reading of a forward and a backward reference leads to the state shown in Figure 7.9

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

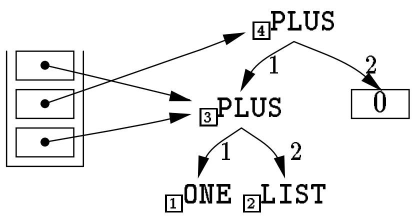


```
read c <RET>
     # 1 <RET>
     b <SPACE> 2 <RET>
     * 1 <RET>
with next 4,
        stack push(5, push(3, push(3, empty))),
             abbreviations [abbr c of type PLUS with sig 2,
                          abbr b of type LIST with sig *,
                          abbr a of type ONE with sig 0]
and nodes [node 1 of type ONE with sig 0 and edges {},
          node 2 of type LIST with sig * and edges {},
          node 3 of type PLUS with sig 2
          and edges {1 → 1,
                     2 → 2}]
```

Figure 7.9 The GEL machine after 3 abbreviations, 3 builds and 2 copies

Forward references are depicted as rectangular boxes containing the number of nodes to be built before the actual node will be built. Therefore, reading two more lines, we get the state displayed in figure 7.10.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

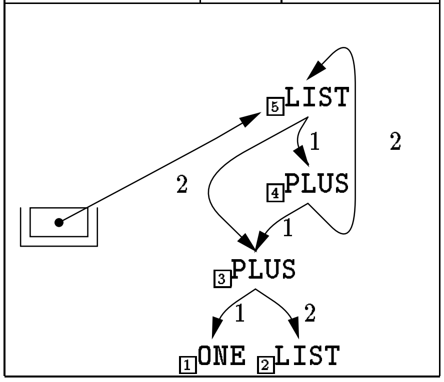


```
read b <SPACE> 2 <RET>
     * 1 <RET>
with next 5,
        stack push(3, push(4, push(3, empty))),
             abbreviations [abbr c of type PLUS with sig 2,
                          abbr b of type LIST with sig *,
                          abbr a of type ONE with sig 0]
and nodes [node 1 of type ONE with sig 0 and edges {},
          node 2 of type LIST with sig * and edges {},
          node 3 of type PLUS with sig 2
          and edges {1 → 1,
                     2 → 2},
          node 4 of type PLUS with sig 2
          and edges {1 → 3,
                     2 → 5}]
```

Figure 7.10 The GEL machine after 3 abbreviations, 3 builds and 2 copies, a build and one more copy

Note, that the future has come one build-step closer; the `right` edge of the `PLUS` node now refers to the next node that will be built.

SHORT-TYPE	SIG	TYPE
a	0	`ONE`
b	-1	`LIST`
c	2	`PLUS`



read
with next 6,
stack push(5, *empty*),
abbreviations [*abbr* c *of type* `PLUS` *with sig* 2,
abbr b *of type* `LIST` *with sig* ∗,
abbr a *of type* `ONE` *with sig* 0]
and nodes [*node* 1 *of type* `ONE` *with sig* 0 *and edges* {},
node 2 *of type* `LIST` *with sig* ∗ *and edges* {},
node 3 *of type* `PLUS` *with sig* 2
and edges {1 → 1,
2 → 2},
node 4 *of type* `PLUS` *with sig* 2
and edges {1 → 3,
2 → 5},
node 5 *of type* `LIST` *with sig* ∗
and edges {1 → 4,
2 → 3}]

Figure 7.11: The GEL machine after 3 abbreviations, 3 builds and 2 copies, a build, one more copy and a drop

7.7.6 Effects of the drop command

The meaning of the drop command is specified in the equation

$$\begin{array}{l} \textit{read} * X\ \textit{Ret} \\ \quad \textit{Gel-Item}^* \\ \textit{with next } X_1,\ \textit{stack Stack},\ \textit{abbreviations Table and nodes } [\textit{Node}^*] = \qquad \text{[read-drop]} \\ \textit{read Gel-Item}^*\ \textit{with next } X_1,\ \textit{stack drop}(X,\ \textit{Stack}),\ \textit{abbreviations Table and nodes } [\textit{Node}^*] \end{array}$$

The result of reading the last two lines in the example is shown in figure 7.11. Note that the GEL text is erroneous if more than one element is left on the stack after the text is processed.

7.8 Evaluation of the specification

We would like to note the following about the specification of GEL reading above.

- The specification should not leave any doubts concerning the mapping of a GEL text into a graph.
- The mapping is overspecified. Firstly, any consistent renumbering of nodes should be considered the same graph. Secondly, for an implementation that *reads* a graph without the need to *write* exactly that graph, there is no need to keep track of subgraphs

after their root disappears from the stack, because they cannot be referenced by any GEL command. E.g., the C library described in the appendix obtains (a pointer to a) function for building a graph from other graphs, which is provided by the user of the library. This funtion returns a result which is never inspected by the functions in the C library itself, so the user of the library has freedom to return as result some *interpretation* of the graph (see also Section 7.10). In this scheme, it is of course required that the user is able to build interpretations of graphs from interpretations of graphs.

- The lexical syntax of GEL could not be specified satisfyingly. It is impossible to have carriage returns occur both in the definition of LAYOUT and in the meaningful part of the syntax.

7.9 A GEL writing algorithm

In the previous sections, we have presented a mapping from GEL texts to graphs (the function *read*). This mapping is an exact definition of the meaning of GEL texts. In general, many GEL texts are mapped (by *read*) to the same graph. Therefore, there is no unique inverse function of *read*. However, given a graph, there is always at least one GEL text that describes it.

Here we give the pseudo code implementing a function *write* that satisfies the requirement that for all graphs g, g=`read`(`write`(g)). The pseudo code is equivalent to the algorithm implemented in the library described in Appendix B.1. Apart from the correctness requirement above, this algorithm satisfies the following requirements:

- Forward references are only generated for edges pointing to ancestors in the depth-first tree associated with the graph (see [CLR89] for this terminology). Given the fixed order in which children are visited, the number of forward references is minimal.
- Nodes with a single parent are written in the order they are needed by their parent, no copy commands are generated for them.

For trees, this algorithm uses no more stack space than the total number of nodes. For balanced trees, it uses only logarithmic stack space. If less than 250 different types occur in an instance, the length of a GEL text asymptotically approximates one character per node when the number of edges is comparable to the number of nodes. The algorithm is divided in three phases:

Phase1 A depth-first traversal is done to detect which nodes are shared. Every node is ordered into a sequence after all its edges have been explored.

Phase2 For every shared node, it is determined how many 'local' nodes are visited in a depth-first traversal starting at the shared node, avoiding subgraphs accessible through shared nodes.

Phase3 The subgraphs associated with the shared nodes are written in the sequence obtained in Phase1. Stack-offsets for shared nodes that have already been written are determined by a simple model of the stack-layout of the GEL-reader, offsets for shared nodes that will be written in the future are determined by the number of intervening 'local' nodes (see Phase2). Finally, if the root node is not a shared node, the subgraph associated with the root node is written.

In pseudo code, the algorithm reads as follows. Global variables are *shared, entered, sequence, onstack, write_seq, written* and *abbreviated.* Procedure Phase1 is called on the root of the graph, procedure Phase2 and Phase3 are called without arguments. In the pseudo code, "*string1<exp>string2*" denotes the string obtained by concatenating *string1*, the (string) value of *exp*, and *string2*.

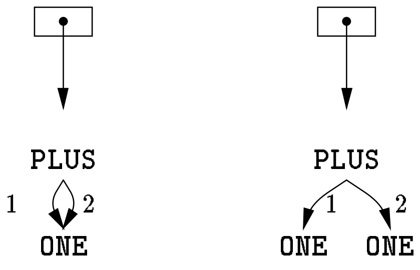
```
shared = {}              % set of shared nodes
entered = {}             % set of entered nodes
sequence = []            % visit order of nodes
onstack = 0              % number of refs on reader stack
next_build = 1           % number of next build
written = {}             % set of nodes already written
abbreviated = {}         % set of types with abbreviation

extern varyadic(n)       % test if node n is varyadic
extern sig(t)            % signature of type t
extern abbr(t)           % unique abbreviation for type t
extern type(n)           % type of node n
extern size(l)           % number of elements in set or list
extern children(n)       % list of edges emanating from node n

phase1(n) {
   if (($n \in entered$) and not ($n \in shared$))
   then $shared = shared \cup \{n\}$
   else
      $entered = entered \cup \{n\}$
      for r in children(n): phase1(r)
      sequence = sequence :: n
   fi
}

int locals(n) { % auxiliary for phase 2
   childrensum = 0
   for r in children(n):
        if ($r \notin shared$)
        then childrensum = childrensum + locals(r)
        fi
   return childrensum + 1
}

phase2() {
   cumlocs = 0
   sharedno = 1
   for r in sequence
```

```
        if ($r \notin shared$)
            $cumlocs = cumlocs + \mathrm{locals}(r)$
            $r.\mathrm{cum} = cumlocs$
            $r.\mathrm{sharedno} = sharedno$
            $sharedno = sharedno + 1$
        fi
}

write($n$) { % auxiliary for phase3
    for $r$ in children($n$)
        if ($r \in shared$)
        then if ($r \in written$)
               then print "# <onstack-$n$.sharedno> \$n$"
               else print "> <$n$.cum-next_build> \$n$"
               fi
        $onstack = onstack + 1$
        else  write($r$)
        fi
    if (type($n$) $\notin abbreviated$)
    then
        print "!<abbr(type($n$))>:<sig(type($n$))>=<type($n$)> \$n$"
        $abbreviated = abbreviated \cup \{\mathrm{type}(n)\}$
    fi
    print "<abbr(type($n$))>"
    if varyadic($n$) then print "␣ <size($n$)> \$n$" else print "\$n$" fi
    $next\_build = next\_build + 1$
    $onstack = onstack$ - (size($n$)-1)
}

phase3() {
for $r$ in $shared$ write($r$)
if $root \notin shared$
then
      write($root$)
      print "* <number($shared$)-1> \$n$"
else
      print "* <number($shared$)\$n$"
fi
}
```

7.10 GEL writing modulo unraveling

In the preceding section, we have shown a correct `write` function, yielding the identity when composed with the `read` function. In some cases, however, it is sufficient if the composition of `read` and `write` yields a graph in the same equivalence class as the input, given an equivalence relation on the graphs.

Examples are term and graph rewriting, where (term) graphs are equivalent if they are in the rewrite relation. Therefore, in a graph rewriting implementation that uses GEL,



(a) (b)

Figure 7.12: Two graphs with the same unravelling

there is no need to build or write the graph as specified in the `read` specification, any equivalent graph (e.g. the normal form) will do.

Also, several graphs may represent the same term (sharing). Two graphs that are nontrivially equivalent in this sense are shown in Figure 7.12.
In graph (a), the node with type `ONE` is shared by two edges, whereas in graph (b), both edges lead to a private copy. Depending on the context, both the introduction of sharing and the removal of sharing can be advantageous to the size of the GEL text. Below, the GEL texts for graphs (a) and (b) are shown:

! a : 0 = ONE <RET>	! a : 0 = ONE <RET>
a <RET>	a <RET>
# 0 <RET>	a <RET>
! b : 2 = PLUS <RET>	! b : 2 = PLUS <RET>
b <RET>	b <RET>

In this example, the expression of the sharing of a single node takes more space than simply writing out two copies (line 3 of the left GEL text takes at least two bytes in the binary encoding, whereas line 3 of the right text takes only one byte). When larger graphs are shared, the reverse becomes true.

Formally, two graphs g_1 and g_2 represent the same term (up to isomorphism) if they have the same *unraveling*. Following [KKSdV93], we first define paths.

> A path in a graph g is a finite or infinite sequence $a, i, b, j, \ldots$ of alternating nodes and integers, beginning and (if finite) ending with a node of g, such that for each m, i, n in the sequence, where m and n are nodes, n is the ith successor of m. If the path starts from a node m and ends at a node n, it is said to be a path from m to n.

Given the notion of paths, the *unraveling* $U(g)$ of a graph g is defined as:

> The unraveling $U(g)$ of a graph g is the term representation of the following forest. The nodes of $U(g)$ are the paths of g which start from the root. Given a node $a, i, b, j, \ldots, y$ of $U(g)$, then this node has the same type as y, and

its successors are all paths of the form $a, i, b, j, \ldots, y, n, z$, where z is the nth successor of y in g.

This notion is defined without reference to peculiarities of a particular application of GEL, and therefore, the library implementation of GEL can exploit it. This does not imply that graphs with the same unraveling are considered equivalent in all applications of GEL, and therefore writing modulo unraveling is parameterized by two predicates on nodes, `unshare` and `compress`. If both predicates are false on all nodes, $\texttt{read} \circ \texttt{write}$ is the identity. If `unshare` is true of some node, multiple edges to a subgraph are unraveled by $\texttt{read} \circ \texttt{write}$ into edges to graphs with the same unraveling. If `compress` is true of some node, multiple edges to a subgraph in the image correspond to edges to some subgraphs with the same unraveling in the origin. In a table:

$\exists i : \texttt{unshare}(i)$	$\forall i : \texttt{not}(\texttt{unshare}(i) \vee \texttt{compress}(i))$	$\exists i : \texttt{compress}(i)$
$i_1 = i_2 \Rightarrow o_1 \simeq o_2$	$i_1 = i_2 \Leftrightarrow o_1 = o_2$	$i_1 \simeq i_2 \Leftarrow o_1 = o_2$

Where a subscripted i refers to a subgraph in the input, a subscripted o refers to the corresponding subgraph in the output, and $g_1 \simeq g_2$ means that g_1 and g_2 have the same unraveling. Note that the use of unraveling equivalence is enabled, but not forced by the parameters. This is because determining if graphs have the same unraveling may be far too expensive.

7.11 Discussion of design decisions

At first sight, there seems no need for the *definition* of an abbreviation to occur in a GEL text. Processing the definition of an abbreviation takes time, so one might argue that if the reader and writer agree in advance on the full names of a `TYPE`, they might as well agree on their abbreviations.

However, there are four good reasons for GEL's abbreviation scheme:

1. A scheme with fixed abbreviations deteriorates when there are many readers and writers using different sets of types. Then, a globally consistent set of abbreviations must be found, resulting in longer abbreviations, and awkward recompilations if new components are added to the system.

2. In GEL's scheme, the number of bits occupied by an abbreviation is determined by the number of `TYPE`s actually occurring in a single graph. This is typically one or two orders of magnitude smaller than the total number of `TYPE`s known in the system. If the number of actually occurring `TYPE`s is still large, the redefinability of abbreviations may be used to keep the `SHORT-TYPE`s small.

3. The abbreviation mechanism is convenient for the exchange of 'external' types like bitmapped pictures.

4. GEL texts are completely self-describing, enabling inspection with tools that are independent of the actual application.

It should be noted that GEL's abbreviation scheme deteriorates to a worst case when every node in the graph has a unique `TYPE`. In the systems we know, this happens only in the case of very small graphs (such as the examples given at the start of this paper), where efficiency is not a severe problem. Even in this worst case, the overhead is only 4 bytes per node.

From the viewpoint of efficiency, one might also argue that the arity should not be specified, because it could be a function of the `TYPE`, known to both reader and writer. However, the specification of arities allows decoding of GEL texts without interpreting `TYPE`s. It should be noted that the arity is specified only once for every type, thus causing a relatively small loss in efficiency.

From a minimalistic point of view, only varyadic arities are needed, because any graph structure can be specified with varyadic types. The introduction of fixed arities, however, allows shorter encodings because the actual number of edges need not be specified for every node.

The copy commands use indices relative to the stack (for backward references) and relative to the building sequence (for forward references). In the writer, this is more difficult to implement than a label scheme as used in, e.g., the ERL of IDL. However,

- Labels have definition and use occurrences, whereas relative indices have only use occurrences. Thus, relative indices yield a shorter representation.
- Stack references by number are cheaper to implement in the reader than label references, forward references are probably equally expensive as label references.
- Relative indices permit compositionality of term graphs; for a term C[t1,t2], the GEL text is the concatenation of the GEL texts for t1, t2 and C. No relabelings are needed. It might be interesting to investigate the support of more powerful graph compositions.
- Even if writing is more expensive, it is to be expected that a GEL representation will be read at least as many times as it is written.

7.12 Measurements

For GEL texts describing graphs of several sizes, we have made measurements of reading and writing time (sum of system time and cpu time), and the number of bytes per node (in the binary representation). The measurements were performed on a Sillicon Graphics Indigo, with a MIPS R4000 processor running at 100 Mhz internal clock frequency. We used the C library presented in Appendix B.1 for the measurements of reading and writing times.

For the sake of comparison, in the last row of the tables, figures are given for a fast, application specific binary writer and reader of binary trees. These programs use fixed codes for the set of types occurring in their trees, and read or print the codes as they are encountered in a preorder traversal. This format is widely known as 'Polish' notation, therefore we have called the programs 'Polish'. Finally, in the last column of the tables, we show the ratio between the GEL functions and the 'Polish' programs.

# nodes	bytes	reading time	ratio
23	17	60.5 ± 0.5	19.8
4471	1.49	6.76 ± 0.35	2.21
68947	1.24	6.75 ± 0.04	2.21
Polish (4095)	1	3.06 ± 0.01	1.00

Table 1. GEL reading time (in μs) per node

# nodes	write (cpu+sys)	ratio
23	723 ± 25	520
4471	14.2 ± 0.8	10.1
68947	17.9 ± 1.0	12.7
Polish (4095)	1.41 ± 0.04	1.00

Table 2. GEL writing time (in μs) per node

In the second column of table 1, we see that the number of bytes per node is high (17) for small graphs, but tends to 1 byte per node for large graphs. Related to this, the processing time per node is much higher for small graphs than for large graphs. We observe that writing is between 2 and 3 times as expensive as reading. Only a small part of the difference can be attributed to the fact that writing a file is more expensive than reading it. It seems more likely that the sharing analysis is expensive. Some more tuning and profiling might reduce the gap between reading and writing.

It is interesting to compare these figures to the figures found by Lamb in his Ph.D. thesis [Lam83], for a signature that is comparable to ours. We have multiplied Lamb's number of bytes per node with 7/8, because Lamb used a 7-bit machine (running at about 2 MIPS). Lamb tested an ASCII writer, a binary writer (binary encodings are given for the node types), a hand-coded dedicated Polish tree-writer (in a pre-order traversal, only a code for the node type is written), and the LG package of the PQCC project [N$^+$78]. The sizes of his structures were between 400 nodes and 10000 nodes, and the values reported are the result of applying linear regression techniques. This implies that Lamb's software is not very sensitive to the size of the graphs.

Package	bytes/node	Input (ms)	ratio
ASCII	14.72	2.38 ± 0.003	21
Binary	12.18	$0.981 \pm .002$	8.8
Polish	2.61	0.111 ± 0.0002	1.0
PQCC	25.43	2.028 ± 0.007	18

Table 3. Lamb's readers

For larger graphs, in terms of the number of bytes per node, our GEL implementation still scores about twice as good as Lamb's Polish writer. This can only be explained if Lamb's codes for the node types are 2.61 bytes long on average. Even for small graphs, GEL does not perform significantly worse than any of the other three packages.

Package	Output (ms)	ratio
ASCII	1.434 ± 0.002	28
Binary	0.360 ± 0.002	7.1
Polish	0.051 ± 0.0003	1.0
PQCC	1.322 ± 0.010	26

Table 4. Lamb's writers

With regard to the processing times, we will only make relative comparisons. That is, we will compare the ratios to the 'polish' readers and writers. First we will consider 'average' terms, of about 4000 nodes. Lamb's ASCII reader performs 21.4 times as bad as his polish reader, whereas our GEL reader performs only 2.2 times as bad as our polish reader. Lamb's ASCII writer performs 28.1 times as bad as his polish writer, whereas our GEL writer performs only 10 times as bad as our polish writer. We attribute this to the fact that the GEL formalism is much closer to polish notation than ERL.

We draw the conclusion that it is hardly ever worthwhile to hand-code a GEL reader for a production version, but hand-coding a GEL writer can be useful. Hand-coding has the additional advantage that maximal knowledge about sharing can be exploited. Of course, if the internal graph representation is such that almost uninterpreted memory dumps can be made, and both reader and writer use the same representation, techniques as described in [New87] can be used.

7.13 Conclusions

GEL is a formalism for the implementation-language independent exchange of graph-structured data. GEL is exclusively concerned with graph-structure, the type of a node is a sequence of bytes, uninterpreted by GEL. The formal semantics of GEL allows an easily verifiable implementation of GEL readers and writers.

GEL is compositional. Especially in generated distributed environments, this is important. There, it often happens that input graphs must be composed of several output graphs.

Asymptotically, GEL representations of large, treelike graphs tend to require only one byte storage for representing one node in the graph.

The speed and compactness of GEL should, in almost all cases, overcome the need for alternative implementations for production versions of tools.

Special thanks go to Job Ganzevoort, Steven Klusener, Paul Klint and Pum Walters, for numerous suggestions and discussions. Job implemented the initial version of the `C` library, Steven turned the formal specification of GEL inside out. Remaining errors are of course the responsibility of the author.

Chapter 8

Assessment

In this final chapter, we assess what has been achieved, and discuss directions for future research.

8.1 Conceptual Achievements

In Chapter 3, we have presented a subclass of Term Rewriting Systems, Minimal Term Rewriting Systems (MTRSs). This class serves as the machine language of an extremely simple abstract machine, the Abstract Rewriting Machine (ARM).

ARM is easy to implement on stock hardware, because each instruction can be mapped to a short sequence of conventional machine instructions. The semantics of ARM is a mere half page of algebraic specification, and it is easy to see that, given an MTRS, ARM implements rightmost innermost term rewriting with specificity ordering of the MTRS.

With respect to other implementations of term and graph rewriting, the most distinguishing feature of ARM is the fact that only normal forms are represented in (expensive) heap space, whereas all unevaluated parts of terms are represented on (cheap) stack space. This is a consequence of the design decision to implement innermost rewriting.

In Chapter 3, we have also introduced a transformation from left-linear TRSs into MTRSs, which can be used as a compiler for TRSs. In earlier discussions of compilers for TRSs, we have not found a comparable combination of concern for simplicity, correctness and efficiency.

In order to actually use the techniques of Chapter 3, a formally well-defined language for the expression of TRSs, and an implementation of the transformation from TRSs into MTRSs are needed. The language EPIC, defined in Chapter 4, can be used to express TRSs, and the transformation has been expressed in EPIC itself in Chapter 5. Unfortunately, this expression of the transformation leaves to be desired with respect to readability.

In Chapter 6, we have defined *lazy rewriting*, analogous to *lazy evaluation* in lazy functional programming languages. Our definition is of a less operational character than that of lazy evaluation in functional programming languages. That is, neither an ordering of rules, nor an ordering of pattern-matching needs to be assumed.

In the same chapter, we have also presented a method to implement lazy rewriting on eager (innermost) machinery. The main achievement of this method is the fact that the efficiency penalty of lazy evaluation (closure building) is only paid when the feature is actually used, i.e., the efficiency of eager rewriting is not influenced at all.

Both the definition of lazy rewriting and the transformation for lazy rewriting on eager machinery further the understanding of the interaction between lazy evaluation and pattern matching.

Finally, in Chapter 7, we define a language for compressed, sharing-preserving, exchange of graphs between distributed software components. This language can be used to combine graph-processing components into a complex, heterogenous system. The measurements in that chapter prove that exchange is fast and compact.

8.2 Practical Achievements

Ideally, we would like to provide here an extensive comparison between EPIC/ARM and other combinations of languages with abstract machines, supported by vast experimental data. For several reasons, however, we have to limit our ambitions in this area.

In the first place, we would like to compare the *design* of abstract machines, rather than their concrete implementations. In such a comparison, the (abstract) efficiency of a design would be expressed in terms of the cost of elementary events occurring in (micro)processors, of conventional or revolutionary construction. Then, given the cost parameters of an existing or proposed concrete machine, one would be able to estimate the performance of the various abstract machine designs.

As already noted in [PJ87, JL92], it is extremely hard to make such comparisons, and so far, few people could be convinced by comparisons of this kind. For the pure lambda-calculus, though, a brave attempt is being made in [DF95]. We think it is worthwhile to extend this approach towards calculi dealing with pattern-matching and other extensions, but the subject is clearly too large to deal with in this thesis.

Confronted with the incomparability on an abstract level, one is forced to compare concrete implementations of abstract machines. This leads immediately to two new problems: Not all abstract machines are implemented on the same concrete machine, and even when this is the case for a particular concrete machine, the results of such a comparison often predict little or nothing about the results on other concrete machines.

Simply taking this problem for granted, and choosing a fixed concrete machine, leads us to the next problem: What problems or programs should be used for testing? The most pure base for comparison would be the construction of the 'best' program for solving a certain, sufficiently general, problem. If a way is found to exclude those programs that simply print the solution (which was computed in some compilation phase), one will find that, except for extremely simple problems, it is impossible to determine which program is the best.

Another poor man's choice leads to considering a class of (restrictions of) programming languages (with their abstract machines) which differ only in concrete syntax, so essentially the same program can be tested. In order to judge the practical value of such a test, the program itself should be practically applicable and have the size of a real-world program. In Section 8.2.1, we summarize the results pertaining to EPIC/ARM of such an experiment.

Given our initial wish to establish the value of the ARM machine model, it is unsatisfactory that the figures obtained in this experiment pertain to a particular combination of an ARM interpreter, a compiler from EPIC (or rather a precursor) to ARM, and a compiler from a subset of ML into EPIC. Furthermore, the core of the computations is formed by floating-point operations, which have been added as an external data type, and are not integrated in the ARM machine model in any way.

The strength of ARM lies in symbolic computations, and in particular in the fact that only normal forms are represented in (expensive) heap space, and all other terms are represented on stacks. Therefore, in Section 8.2.2, we present a tiny problem that exercises precisely this part of ARM, and we compare the implementation tested in Section 8.2.1 with an implementation that would use the compilation algorithm presented in this thesis, and a mapping to machine code obtained from a C program. For reference, the Clean system (the fastest system from the benchmark in Section 8.2.1) is also incorporated in this test.

8.2.1 The Pseudoknot Benchmark

At the Dagstuhl Workshop on Functional Programming in the Real World in May 1994 ([GH94]), a number of workshop participants decided to test their compiler technology on a problem from molecular biology, the Pseudoknot problem [MMG94]. Programs solving this problem had already been written in several languages, including C [KR78], Scheme [RC91], Multilisp [Jr.85], and Miranda (Miranda is a trademark of Research Software Ltd.) [Tur85].

After the workshop, the work on improving both the compilation and execution speed of the Pseudoknot programs continued, and several researchers, including us, joined the team. In [HF$^+$96], the result of the entire enterprise is published. Here, we would like to summarize the results pertaining to EPIC. Two versions of EPIC/ARM were used for the benchmark, referred to as EPIC and EPIC-C. The difference between EPIC and EPIC-C is as follows.

An interesting feature of the ARM interpreter is that individual functions can be overruled by implementations in C, by linking the corresponding object code to the ARM interpreter. This leads to a stratum of possibilities, with, in one extreme, all functions being interpreted through ARM code, and in the other, all functions being implemented in C. With EPIC, we refer to a version of the ARM interpreter in which only floating-point operations were implemented in C, and with EPIC-C, we refer to a version in which for every function of the pseudoknot program, the ARM code was compiled into a C function by a naive translator.

compiler	route	space	pseudoknots
Compiled via C, L(isp) or S(cheme)			
Bigloo	C	7.5	53
Camloo	S+C	4.6	31
Sisal	C	2.4	12
Gambit	C	8.7	10
Yale	L	14	8
ΓCMC	C	13	8
FAST	C	100	6
Opal	C	15	2
Glasgow	C	47	2
Erlang BEAM	C	8	< 0.7
CeML	C	35	< 0.7
ID	C	64	< 0.7
Epic-C	C	**12.4**	< 0.4
Stoffel	C	25	< 0.4
Compiled into native code			
Clean	N	9	90
RUFL	N	3	65
CMU CL	N	14	40
Caml Gallium	N	3.8	31
SML/NJ	N	35	22
LML Chalmers	N	14.2	17
LML(OP-TIM)	N	13.6	15
Facile	N	11.3	14
MLWorks	N	14.4	12
Chalmers	N	50	7
Interpreted			
Gofer	I	3	403
RUFLI	I	1	297
Miranda	I	13	216
Caml Light	I	2.3	91
Trafola	I	6	86
Epic	**I**	**8.4**	24
NHC(HBC)	I	30	13
NHC(NHC)	I	8.7	3
C compilers			
SUN CC -O	N	8	8
GNU GCC -O	N	21	3

Table 8.1. Memory and Time requirements for the Compilation of Pseudoknot

The two most relevant figures are compilation time and execution time. In Table 8.1, the compilation times for the various implementations are given. In order to compensate for the fact that the compilers ran on 19 (!) different machines, compilation time was expressed in pseudoknots, a dimensionless unit which was computed as follows:

$$\text{relative speed (pseudoknots)} = \frac{1000 * \text{ execution time of C version (seconds)}}{\text{compilation time (seconds)}}$$

In [HF$^+$96], both user and system time are expressed as pseudoknots, but because user time dominated system time by a factor of at least 2.73 in all cases, we only reproduce the pseudoknots for user time here.

With regard to the compilation time of Epic, one finds that most of the interpreted systems, but few of the native code compilers, and few of the compilers using another high level language, compile faster than Epic. We can conclude from this, that the Epic compiler is rather slow. However, it should be kept in mind that the Epic compiler is written in Epic itself, and is run by an interpreter. Compared to the other two bootstrapping compilers, NHC(HBC) and NHC(NHC), it is doing rather well.

The compilation time of Epic-C is simply disappointing. This is partly explained by the fact that the generated C code is a faithful mimicking of the interpreter steps.

With regard to the memory consumption of the Epic compiler, one sees computes that half of the interpreted systems, and one third of the native code compilers and compilers using another high level language use less memory for compilation than the Epic compiler. Taking into account that Epic is a bootstrapped compiler, this is a nice result.

Unlike the compilation time, the memory consumption of Epic-C is acceptable, and is in the same range as that of the C compilers themselves.

For the execution times, see Table 8.2. It is easily seen that Epic shares the first position of the interpreted systems with Caml Light, again a nice result. With regard to the compiled systems, we note that only RUFL is slower than Epic-C.

The latter result can be attributed to two factors. Firstly, the C code is obtained by translating all individual ARM instructions to C statements, no effort is done to combine the effect of a sequence of ARM instructions into a more efficient sequence of C statements.

Secondly, both Epic and Epic-C treat the floating point numbers as 'external datatypes', which have a boxed representation, i.e. every use of a floating point number involves a pointer dereference and every floating point result involves memory allocation, which influences the performance negatively. The enormous amount of memory allocated indirectly stems from dealing with the floating point numbers as 'external datatypes'; at the time of the benchmark, there was not yet garbage collection over external datatypes.

compiler	route	precision	time (s)		space
			user	sys	Mb
Compiled via C, L(isp) or S(cheme)					
Glasgow	C	single	3.9	+ 0.2	1
Opal	C	single	4.7	+ 0.5	0.8
CeML	C	single	8.7	+ 0.6	2
FAST	C	single	11.0	+ 0.5	1
Yale	L	single	11.9	+ 7.2	14
Epic-C	**C**	**single**	**43.9**	+ **2.9**	**23**
Sisal	C	double	3.7	+ 0.2	0.7
Gambit	C	double	6.2	+ 0.7	4.4
Camloo	S+C	double	11.2	+ 1.5	4.9
ID	C	double	11.6	+ 2.9	14
Bigloo	C	double	11.7	+ 2.4	4.9
ΓCMC	C	double	14.7	+ 1.1	22
Stoffel	C	double	26.6	+ 2.1	5.6
Erlang BEAM	C	double	31.8	+ 4.5	11
Compiled into native code					
CMU CL	N	single	5.8	+ 3.3	14
LML(OP-TIM)	N	single	7.7	+ 0.3	1.2
Chalmers	N	single	12.1	+ 1.0	3
LML Chalmers	N	single	12.5	+ 0.4	2.1
Caml Gallium	N	double	5.1	+ 0.5	0.3
Clean	N	double	5.1	+ 0.8	2.5
MLWorks	N	double	6.3	+ 0.1	0.3
SML/NJ	N	double	6.9	+ 1.2	2.6
Facile	N	double	15.5	+ 4.3	7.9
RUFL	N	double	87	+ 2.8	3
Interpreted					
Epic	**I**	**single**	**56**	+ **2.8**	**21**
Trafola	I	single	124	+ 6.3	10.7
NHC	I	single	176	+ 5.7	2.6
Gofer	I	single	370	+12.0	3
Caml Light	I	double	52	+ 7.4	0.3
RUFLI	I	double	529	+13.0	4
Miranda	I	double	1156	+34	13
C compilers					
GNU GCC	C	single	2.4	+ 0.1	0.3
GNU GCC	C	double	2.7	+ 0.1	0.3

Table 8.2. Execution times for Pseudoknot

8.2.2 What is the Top Speed?

As signalled above, it is unsatisfactory that the results in the previous section pertain to a compiler that differs from the one presented in this thesis (with one optimization, the compiler in this thesis should improve on that compiler), and to floating-point operations, which are not even built-in to ARM, and irrelevant to term rewriting in general.

Here, we discuss a tiny program, which is designed to test the main strong point of ARM; the fact that only normal forms are represented in (expensive) heap space, and all other terms are represented on stacks. Specifically, the test is designed to avoid the consequences of the following implementation choices:

- How to implement space allocation for a new term?
- What garbage collection algorithm to choose?
- How to implement tests for heap and stack overflow?
- How to represent terms headed by a constant?
- How to represent terms with arguments?
- How to implement pattern matching?
- Should pointers be tagged to distinguish between types?

Also, we want to avoid the consequences of hardware factors such as

- caching schemes;
- number and kind of registers;
- support for calling conventions.

Here, we present a program that can be used as an objective measure for the *maximum* speed that can possibly be reached by ARM technology on a certain hardware platform. We would like to stress again that this number is of limited use. It may be used to assess the *relative* impact of implementation decisions, or as a *very* rough comparison between language implementations, but it is not a reliable performance estimator for real-world problems. In this sense it is reminiscent of the so-called `nfib` benchmark. Because our program does not use machine arithmetic, we prefer it over the `nfib` benchmark as a measure of raw speed.

In order to eliminate the factors mentioned above, this program has the following properties:

- For innermost evaluation, no heap space needs to be allocated at all.
- The number of reductions performed is 2^k, where k is the number of functions defined in the program, so it is easy to test a large number of reductions.

$$
\begin{array}{lcl}
\textbf{module} & F & \\
\textbf{types} & & \\
\quad f_0 & : & \to T; \\
\quad f_1 & : & T \to T; \\
\quad f_2 & : & T \to T; \\
\quad \ldots & & \\
\quad f_{k-1} & : & T \to T; \\
\quad f_k & : & T \to T; \\
\textbf{rules} & & \\
\quad f_1(X) & = & X; \\
\quad f_2(X) & = & f_1(f_1(X)); \\
\quad \ldots & & \\
\quad f_k(X) & = & f_{k-1}(f_{k-1}(X));
\end{array}
$$

Figure 8.1: A program for measuring maximum speed

- The program does not perform pattern matching.
- The program does not exploit machine arithmetic or call external functions.
- The stackdepth is limited to k. On machines with a cache, this should cause the part of the stack that is used to stay entirely in the cache.
- The machine code should be representable in ck words, where c is a small constant. As in the previous point, this should cause the code to remain entirely in the cache.
- All functions have a single argument, which is passed on unchanged. Therefore, register allocation (if any) should be extremely simple.

The EPIC version of this program is displayed in Figure 8.1.

It is easily verified, that reduction of the term $f_k(f_0)$ takes 2^k steps, yielding f_0 as normal form. We will express the *maximum speed* of any implementation as the number of reductions per second for this program.

For $k = 24$, the ARM program resulting from compilation of the EPIC program above (using the algorithm described in this thesis) is given in Figure 8.2. It should be noted that the ARM program produced by the compiler tested in Section 8.2.1 contains some superfluous movements from the A stack to the traversal stack and back.

In Figure 8.3, a manually produced C implementation of the ARM code in Figure 8.2 is shown. The program uses a GNU language extension which allows the address of a label to be passed on in a variable: With `v=&&l`, the address of label `l` is assigned to the variable `v`, so a jump to `l` is caused by the statement `goto **v`. Using `gcc -O` on this program we obtain assembly code that should be close to the assembly code to be produced by a machine-specific implementation of EPIC.

```
f0:     build(f0,0)
        recycle
f1:     recycle
f2:     cpush(f1)
        goto(f1)
f3:     cpush(f2)
        goto(f2)
...
f24:    cpush(f23)
        goto(f23)
```

Figure 8.2: An ARM program for the Epic source in Figure 8.1

```
#define LABEL(F) l_##F:         /* ARM labels are C labels              */
#define CPUSH(F) *++C=&&l_##F;  /* C is C stack ptr, &&l address of l   */
                                /*   stack overflow to be handled by MMU  */
#define GOTO(F)  goto l_##F;    /* ARM goto is C goto                   */
#define STOP     exit(0);       /* exit gracefully                      */
#define RECYCLE  goto **C--;    /* RECYCLE is jump to label popped from C */

main() {
        /* A stacksize of 100 is enough */
        void **C = (void **)malloc(100*sizeof(void *));

        /* push the arguments in preorder */
        CPUSH(stop)
        GOTO(f24)

        LABEL(stop)
          STOP
        LABEL(f1)
          RECYCLE
        LABEL(f2)
          CPUSH(f1)
          GOTO(f1)
        LABEL(f3)
          CPUSH(f2)
          GOTO(f2)

        /* etcetera */

        LABEL(f24)
          CPUSH(f23)
          GOTO(f23)
}
```

Figure 8.3: A (GNU) C representation of the ARM program in Figure 8.2

For comparison, we have expressed the program also in Clean, the fastest compiler producing native code in the pseudoknot benchmark. Clean versions of the program are given in Figure 8.4. Because basic datatypes such as integers receive special treatment in Clean, the left program uses the integer '1' instead of the constructor `F0` used in the right program.

```
module fasti                  module fastb

import StdInt                 ::B = F0

f1 :: !Int -> Int             f1 :: !B -> B
f1 x = x                      f1 x = x

f2 :: !Int -> Int             f2 :: !B -> B
f2 x = f1 (f1 x)              f2 x = f1 (f1 x)

....                          ...

f24 :: !Int -> Int            f24 :: !B -> B
f24 x = f23 (f23 x)           f24 x = f23 (f23 x)

Start :: Int                  Start :: B
Start = f24 1                 Start = f24 F0
```

Figure 8.4: Two Clean programs

We measured the performance of all these programs on the same machine, a PC with an AMD 486DX2-66MHz processor, a 256Kb cache, and 16Mb of internal memory, running Linux kernel version 1.2.8.

The results of these measurements are given in Table 8.1. The measurements are numbered in the first column of this table. For clarity, we provide a condensed description of the versions:

1 This is the Clean program on the left hand side of Figure 8.4. In this program, the integer '1' is passed around, but not inspected.

2 This is the executable obtained by compiling the C program in Figure 8.3. It is meant as an indication of the speed that a machine-specific implementation of ARM would reach.

3 This is the ARM interpreter used in all other benchmarks, but the ARM code was manually changed to conform to the compilation scheme described in this thesis, and the memory checks were left out.

Number	Program	user time (s)	rewrites/second
1	Clean (integer)	1.40	$12 * 10^6$
2	'Machine specific ARM' (GNU C)	1.65	$10.2 * 10^6$
3	ARM interpreter (new compilation scheme, no memory checks)	14.3	$1.2 * 10^6$
4	Clean (F0)	19.0	$0.88 * 10^6$
5	ARM interpreter (new compilation scheme, with memory checks)	37.5	$0.45 * 10^6$
6	ARM interpreter (old compilation scheme, with memory checks)	97.7	$0.17 * 10^6$

Table 8.1: Results on the maximum speed benchmark

4 This is the Clean program on the right hand side of Figure 8.4. In this program, the user-defined constructor 'F0' is passed around, but not inspected.

5 This is the ARM interpreter used in all other benchmarks, but the ARM code was manually changed to conform to the compilation scheme described in this thesis. The memory checks were maintained, however.

6 This is the ARM interpreter used in all other benchmarks, executing the ARM code as generated by the compiler used in Section 8.2.1.

The first noteworthy point is the speed of the integer version of the Clean (integer) program; it even beats the GNU C program we used as a model for the machine code to be generated for ARM. This corroborates the claim of the Clean group that for optimal speed (of both execution and compilation), one should not rely on C compilers.

Somewhat surprising is the dramatic effect of replacing the integer type by a user-defined data type: then the code produced by the Clean *compiler* runs 13.6 times as slow, even worse than the ARM *interpreter* without memory checks.

Switching the memory checks on is clearly very expensive; the ARM interpreter becomes 2.5 times as slow. In a machine- and operating system-specific version, memory checks can be implemented by interrupts from a memory manager, which only occur when the memory is actually exhausted. To date, we have not found a way to solve this problem that is both efficient and portable.

Finally, it is clear that for the program in this section, the new compilation algorithm produces better code than the compiler tested in Section 8.2.1, because the code produced by that compiler is again more than twice as slow.

It should be noted that for realistic examples, the speed attained by the various configurations above is expected to be closer together. Among other factors, this is caused by the fact that, for larger stretches of ARM code, the memory checks do not dominate as much as here, and waiting for access to main memory takes equally long for all configurations.

Finally, we note that typical implementations of term rewriting systems, such as found in the LeLisp implementation of the ASF+SDF system, the Larch Prover, or the OBJ system, have maximum speeds that do not exceed 3000 reductions per second on machines with comparable processors.

8.3 Future Work

As seems to be usual, the new ideas arising during our research by far outnumber the ideas that could be investigated in detail. Here, we would like to mention a few of the most relevant issues.

It appears that the laziness transformation in Chapter 6 could be expressed much more concisely on MTRSs. By specifying how the laziness annotations for newly introduced functions are obtained from original annotations in the compilation from TRSs into MTRSs, a simpler version of the laziness transformation could be obtained. We intend to use this approach in an actual implementation of lazy rewriting.

Using an approach similar to the one in Chapter 6, more intricate strategies, such as the user-defined strategies in OBJ, could be implemented by transformations on TRSs. Some research in this direction was already done in collaboration with Jaco van de Pol, and we intend to continue this work.

In a natural way, the complexity of (innermost) rewriting in MTRSs can be expressed in terms of the costs of ARM instructions, which can be taken to be small constants. Given such a measure, the complexity of rewriting (given a certain strategy) in TRSs can also be defined. A thorough investigation of this subject might enable the introduction of TRSs (and functional programming languages based on TRSs) in the field of time- and space-critical applications.

The compilation algorithm described and proved correct in Chapter 3 produces acceptable code, but it does not implement some obvious optimizations such as common subexpression elimination. It is an interesting project to express these optimizations as transformations on TRSs, and proving them correct.

Finally, in a parallel context, the **C** form of MTRSs could be interpreted as a *fork* of a parallel rewrite process. We intend to perform an experiment in parallel evaluation using this interpretation.

Appendix A

EPIC in EPIC Sources

This appendix is structured as follows. First, in Section A.1, we briefly describe literate programming in *noweb*. Then, in Section A.2, we introduce the signature of basic functions used throughout the specification. Then, in Section A.4, we state the signature of the functions operating on EPIC abstract syntax (the actual implementation is not relevant to this chapter). Based on this abstract syntax, we develop some utility functions in Section A.5, and a representation which is more geared towards MTRSs in Section A.6. Then, in Sections A.7.1, A.7.2 and A.7.3 we present a transformation to make TRSs simply complete, a transformation to make LHSs of non-MTRS rules most general, and a transformation to remove non-compliant RHSs, respectively.

A.1 Literate Programming in NoWeb

This appendix is a literate program, which means that program and documentation are derived from a single source. The program is divided in chunks, whose definition may be distributed over the document. As an example, consider the first part of the chunk *example text*:

127a ⟨*example text* 127a⟩≡

```
    This is example text A
```

The label in the left margin (consisting of the page number, 127, and possibly a letter) can be used to quickly find the definitions of this chunk.

A chunk may be used in the definition of another chunk:

127b ⟨*example.file* 127b⟩≡

⟨*example text* 127a⟩

```
    Above, we have texts A and B
```

This code is written to file `example.file`.

Chunks with names that do not contain spaces are written to files with the same name as the chunk. So, for this example, the file `example.file` will contain the text:

```
This is example text A
This is example text B
Above, we have texts A and B
```

The second line is also part of the chunk ‘example text’, but this part of the chunk is defined later:

128a ⟨*example text* 127a⟩+≡

```
This is example text B
```

A.2 Booleans, Integers and Strings

We use the sort `YN` (short for Yes/No) with constructors `yes` and `no` as a boolean sort with negation `not`.

128b ⟨*yesno esg* 128b⟩≡

```
yes :     -> YN {free};
no  :     -> YN {free};
not : YN -> YN {external};
```

For the specification of the integers, we follow the recipe given in [WZ95]. To write the compiler, we only need the digits 0 and 1, denoted by the functions `d0` and `d1` (though in the output, larger numbers may appear), the unary minus `neg`, the addition function `plus`, the subtraction `min`, and boolean tests `gt` denoting >.

To the specification in [WZ95], we have added `is_nat` for testing for nonnegative integers, the successor function `succ`, and `eq_int` for equality on integers.

128c ⟨*int esg* 128c⟩≡

```
d0      :           -> Int {free};
d1      :           -> Int {free};

succ    : Int       -> Int {external};
plus    : Int # Int -> Int {external};
min     : Int # Int -> Int {external};
gt      : Int # Int -> YN  {external};
is_nat  : Int       -> YN  {external};
eq_int  : Int # Int -> YN  {external};
```

We use strings to construct atomic identifiers (in the next section), using the convention that `'foo`` denotes the string ”foo”. Actually, ‘foo‘ is shorthand notation for the term `str('f,str('o,str('o,eos)))`, where `'f` and `'o` are functions denoting characters, and `str` and `eos` are functions denoting strings. We will not use these concrete functions in our specifications. We *will* use `cat`, a free constructor concatenating two strings.

128d ⟨*char esg* 128d⟩≡

```
'#:            -> Char {free};   '`:            -> Char {free};
```

```
'A:           -> Char {free};   'a:           -> Char {free};
'B:           -> Char {free};   'b:           -> Char {free};
'C:           -> Char {free};   'c:           -> Char {free};
'D:           -> Char {free};   'd:           -> Char {free};
'E:           -> Char {free};   'e:           -> Char {free};
'F:           -> Char {free};   'f:           -> Char {free};
'G:           -> Char {free};   'g:           -> Char {free};
'H:           -> Char {free};   'h:           -> Char {free};
'I:           -> Char {free};   'i:           -> Char {free};
'J:           -> Char {free};   'j:           -> Char {free};
'K:           -> Char {free};   'k:           -> Char {free};
'L:           -> Char {free};   'l:           -> Char {free};
'M:           -> Char {free};   'm:           -> Char {free};
'N:           -> Char {free};   'n:           -> Char {free};
'O:           -> Char {free};   'o:           -> Char {free};
'P:           -> Char {free};   'p:           -> Char {free};
'Q:           -> Char {free};   'q:           -> Char {free};
'R:           -> Char {free};   'r:           -> Char {free};
'S:           -> Char {free};   's:           -> Char {free};
'T:           -> Char {free};   't:           -> Char {free};
'U:           -> Char {free};   'u:           -> Char {free};
'V:           -> Char {free};   'v:           -> Char {free};
'W:           -> Char {free};   'w:           -> Char {free};
'X:           -> Char {free};   'x:           -> Char {free};
'Y:           -> Char {free};   'y:           -> Char {free};
'Z:           -> Char {free};   'z:           -> Char {free};
'[:           -> Char {free};   '{:           -> Char {free};
']:           -> Char {free};   '|:           -> Char {free};
'^:           -> Char {free};   '}:           -> Char {free};
'_:           -> Char {free};   '~:           -> Char {free};

str    : Char # Str             -> Str {free};
eos    :                        -> Str {free};
chr2str: Char                   -> Str {external};
scanchr: Str                    -> Tup {external};
eos    :                        -> Str  {free};
cat    : CharOrStr # CharOrStr  -> Str  {free};
int2str: Int                    -> Char {external};
eq_str : Str # Str              -> YN   {external};
```

A.3 Identifiers

The sort `Id`, denoting identifiers, is a parameter of EPIC abstract syntax.

We will construct atomic function symbols from strings using `mk_id`, we will make the *constructor variant* of a symbol f with `cvar`(f), we will annotate function symbols with other function symbols using `cat_id`, and we will annotate function symbols with natural numbers using `nat_id`.

130a ⟨*id esg* 130a⟩≡

```
mk_id      : Str          -> Id  {external};
cat_id     : Id # Id      -> Id  {external};
nat_id     : Id # Nat     -> Id  {external};
cvar       : Id           -> Id  {external};
```

In running text, the constructor variant of `f` is printed as `"f!"`, given that `F` is printed as `f` and `G` is printed as `g`, `cat_id(F,G)` is printed as `"f#g"`, and natural number annotations are not shown. The textual representations are obtained with the function `id2str`.

130b ⟨*id esg* 130a⟩+≡

```
id2str     : Id # Str     -> Str {external};
id2str2    : Id # Str     -> Str {external};
```

Throughout, the natural number annotation on a function symbol f can be interpreted as the number of arguments of f that resides on the *traversal stack* (the *locus*, see Chapter 3). For symbols without annotation, this number is taken to be 0. In running text, we will denote this number also with $L(f)$.

Consistently with the fact that the locus of symbols without annotation is taken to be 0, an `Id` with annotation `d0` rewrites to the `Id` without annotation. Furthermore, a symbol cannot have two locus annotations, only the 'outermost' or 'last' one counts, and the locus of a root symbol `Id1` that is annotated with another symbol `Id2` is taken to be the locus of the root symbol `Id1`.

130c ⟨*id rules* 130c⟩≡

```
nat_id(Id,d0) = Id;
nat_id(nat_id(Id1,Nat1),Nat2) = nat_id(Id1,Nat2);
cat_id(nat_id(Id,Nat),Id2) = nat_id(cat_id(Id,Id2),Nat);
```

We can obtain the locus annotation with `get_nat`, and we can test for equality (modulo constructor variants) with `eq_id`.

130d ⟨*id esg* 130a⟩+≡

```
get_nat    : Id           -> Nat {external};
eq_id      : Id # Id      -> YN  {external};
```

130e ⟨*id rules* 130c⟩+≡

```
get_nat(cvar(Id)) = d0;
get_nat(mk_id(Str)) = d0;
get_nat(nat_id(Id,Nat)) = Nat;
get_nat(cat_id(Id1,Id2)) = d0;
```

The equality of identifiers is taken modulo constructor variants.

131a ⟨*id rules* 130c⟩+≡

```
eq_id(I1,I2) = no;
eq_id(cvar(Id1),Id2) = eq_id(Id1,Id2);
eq_id(Id1,cvar(Id2)) = eq_id(Id1,Id2);
eq_id(mk_id(S1),mk_id(S2)) = eq_str(S1,S2);
eq_id(cat_id(Id1,Id2),cat_id(Id3,Id4)) =
  given eq_id(Id1,Id3) as
        yes : eq_id(Id2,Id4)
        no  : no;
eq_id(nat_id(Id1,Nat1),nat_id(Id2,Nat2)) =
  given eq_int(Nat1,Nat2) as
        yes : eq_id(Id1,Id2)
        no  : no;
```

A.4 EPIC Abstract Syntax

Here, we give the EPIC names for the functions specified in Chapter 4. Our implementation conforms to that specification, but, because the compiler presented here is *representation independent* with respect to the abstract syntax of EPIC, we omit the actual implementation.

A.4.1 Access functions

An EPIC program consists of a sequence of modules, which is obtained by the function `e_subs_m`. We will describe the compilation of a single module, using the function `e_at_m` to obtain the first module of a sequence of modules.

131b ⟨*epic abstract syntax access* 131b⟩≡

```
e_subs_m    : Prog            -> Sq_m {external};
e_at_m      : Sq_m            -> Mod    {external};
```

A module consists of sequences of signature declarations and rules. The signature declarations are obtained by `e_subs_f`, and `e_at_f` and `e_adv_f` are used to obtain the first declaration of a sequence and the rest, respectively.

A signature declaration has a name, an arity, a result sort, argument sorts and possibly attributes. These parts are obtained by `e_name`, `e_arity`, `e_rsort`, `e_asorts` and `e_attrs`, respectively.

The result and argument sorts are not significant in EPIC. In the abstract syntax, the sorts are represented by terms. This leaves much freedom for interpretation by external tools.

Likewise, the attributes are represented as terms. Two attributes, *free* and *external* play a special role. They can be tested by `e_free` and `e_external`.

131c ⟨*epic abstract syntax access* 131b⟩+≡
```
e_subs_f     : Mod            -> Sq_f {external};
e_at_f       : Sq_f           -> Type {external};
e_adv_f      : Sq_f           -> Sq_f {external};
e_name       : Type           -> Id   {external};
e_arity      : Type           -> Nat  {external};
e_rsort      : Type           -> Term {external};
e_asorts     : Type           -> Sq_t {external};
e_attrs      : Type           -> Sq_t {external};
e_external   : Type           -> YN   {external};
e_free       : Type           -> YN   {external};
```

The rules of a module are obtained by `e_subs_r`; `e_at_r` is used to obtain the first rule or a sequence of rules; and `e_adv_r` yields the rest of a sequence of rules. The LHS of a rule is obtained by `e_lhs`, the RHS by `e_rhs`. Given a term, we can test whether it is a variable with `e_is_var`, we can obtain its outermost function symbol with `e_ofs`, and its subterms with `e_subs_t`. The first term of a sequence of terms is obtained with `e_at_t`, and the rest of a sequence of terms is obtained with `e_adv_t`.

132a ⟨*epic abstract syntax access* 131b⟩+≡
```
e_subs_r     : Mod            -> Sq_r {external};
e_at_r       : Sq_r           -> Rule {external};
e_adv_r      : Sq_r           -> Sq_r {external};
e_lhs        : Rule           -> Term {external};
e_rhs        : RuleOrCase     -> Term {external};
e_is_var     : Term           -> YN   {external};
e_ofs        : Term           -> Id   {external};
e_subs_t     : Term           -> Sq_t {external};
e_at_t       : Sq_t           -> Term {external};
e_adv_t      : Sq_t           -> Sq_t {external};
```

Given any sequence, be it of types, rules, or terms, we can determine whether it contains any elements at all with `e_null`, and we can determine the number of elements with `e_number`.

132b ⟨*epic abstract syntax access* 131b⟩+≡
```
e_null       : Sq             -> YN     {external};
e_number     : Sq             -> Nat    {external};
```

A.4.2 Functions for constructing EPIC abstract syntax

The constructs that are taken apart by the functions in the previous section are put together by the functions below. Their names should be self-explanatory, except maybe `mk_cat`, which is used to concatenate two indices (of the same type, though this is not required by EPIC).

132c ⟨*epic abstract syntax build* 132c⟩≡

```
mk_null      :                     -> Sq   {external};
mk_cat       : Sq # Sq             -> Sq   {external};
mk_var       : Id                  -> Term {external};
mk_type      : Id # Sq_t # Term -> Type {external};
mk_add_attrs: Sq_t # Type          -> Type {external};
mk_indx_t    : Term # Sq_t         -> Sq_t {external};
mk_term      : Id # Sq_t           -> Sq_t {external};
mk_rule      : Term # Term         -> Rule {external};
mk_indx_r    : Rule # Sq_r         -> Sq_r {external};
mk_indx_f    : Type # Sq_f         -> Sq_f {external};
mk_mod       : Sq_t # Sq_r         -> Mod  {external};
mk_prog      : Mod                 -> Prog {external};
```

A.5 EPIC utilities

We also use a library of utility functions on top of the functions provided by the abstract syntax of EPIC.

A.5.1 Finding a type for an Id

Given an `Id` and a sequence of types `Sq_f` containing the type of `Id`, this type is obtained by `type_of`. Note that `type_of` assumes the type of `Id` to occur in `Sq_f`.

133a ⟨*epic-utl esg* 133a⟩≡

```
type_of : Id # Sq_f -> Type {external};
```

133b ⟨*epic-utl rules* 133b⟩≡

```
type_of(Id,Sq_f) =
  given eq_id(Id,e_name(e_at_f(Sq_f))) as
    yes : e_at_f(Sq_f)
    no  : type_of(Id,e_adv_f(Sq_f));
```

A.5.2 Manipulation of Term Indices

Given a sequence $t_0, \ldots, t_n, \ldots, t_m$, `adv_n_t` can be used to obtain $t_n, \ldots, t_m$.

133c ⟨*epic-utl esg* 133a⟩+≡

```
adv_n_t  : Nat # Sq_t -> Sq_t {external};
```

133d ⟨*epic-utl rules* 133b⟩+≡

```
adv_n_t(N,Sq_t) =
  given gt(N,d0) as
    no  : Sq_t
    yes : adv_n_t(min(N,d1),e_adv_t(Sq_t));
```

To obtain t_i $(i \geq 1)$ from a sequence $\vec{t}$, we use **`nth_t`**.

134a ⟨*epic-utl esg* 133a⟩+≡

```
nth_t    : Nat # Sq_t       -> Term {external};
```

134b ⟨*epic-utl rules* 133b⟩+≡

```
nth_t(N,Sq_t) = e_at_t(adv_n_t(min(N,d1),Sq_t));
```

Given a sequence $t_1, \ldots, t_{n-1}, t_n, \ldots, t_m$, the prefix $t_1, \ldots, t_{n-1}$ is obtained by **`pre_n_t`**.

134c ⟨*epic-utl esg* 133a⟩+≡

```
pre_n_t  : Nat # Sq_t       -> Sq_t {external};
```

134d ⟨*epic-utl rules* 133b⟩+≡

```
pre_n_t(N,Sq_t) =
  given gt(N,d1) as
    no  : mk_null
    yes : mk_indx_t(e_at_t(Sq_t),pre_n_t(min(N,d1),e_adv_t(Sq_t)));
```

A sequence is reversed by `rev_t`. We use an auxiliary function `rev_t2` with an accumulator argument. It is easy to see that `rev_t2(Sq,mk_null)` rewrites to the reverse of `Sq`.

134e ⟨*epic-utl esg* 133a⟩+≡

```
rev_t  : Sq_t        -> Sq_t {external};
```

134f ⟨*epic-utl sg* 134f⟩≡

```
rev_t2 : Sq_t # Sq_t -> Sq_t;
```

134g ⟨*epic-utl rules* 133b⟩+≡

```
rev_t(Sq_t) = rev_t2(Sq_t,mk_null);
rev_t2(Sq_t,Sq_t2) =
  given e_null(Sq_t) as
    yes : Sq_t2
    no  : rev_t2(e_adv_t(Sq_t),mk_indx_t(e_at_t(Sq_t),Sq_t2));
```

A.5.3 All variables in a term

134h ⟨*epic-utl esg* 133a⟩+≡

```
vars : Term -> Sq_t {external};
```

134i ⟨*epic-utl sg* 134f⟩+≡

```
vars      : Term -> Sq_t;
vars_args : Sq_t -> Sq_t;
```

135a ⟨*epic-utl rules* 133b⟩+≡

```
vars(Term) =
  given e_is_var(Term),e_subs_t(Term) as
    yes,Args : mk_indx_t(Term,Args)
    no,Args  : vars_args(Args);

vars_args(Args) =
  given e_null(Args) as
    yes : Args
    no  : mk_cat(vars(e_at_t(Args)),vars_args(e_adv_t(Args)));
```

A.5.4 Equality of Terms

We have functions `eq_term` and `eq_terms` for determining equality of terms and indices to terms.

135b ⟨*epic-utl esg* 133a⟩+≡

```
eq_term    : Term # Term -> YN {external};
eq_terms   : Sq_t # Sq_t -> YN {external};
```

135c ⟨*epic-utl rules* 133b⟩+≡

```
eq_term(Term1,Term2) =
  given e_is_var(Term1),e_is_var(Term2) as
    X   ,Y   : no
    yes,yes : eq_id(e_name(Term1),e_name(Term2))
    no ,no  :
      given eq_id(e_ofs(Term1),e_ofs(Term2)) as
        yes : eq_terms(e_subs_t(Term1),e_subs_t(Term2))
        no  : no;

eq_terms(Terms1,Terms2) =
  given e_null(Terms1),e_null(Terms2) as
    X   , Y   : no
    yes, yes : yes
    no , no  :
      given eq_term(e_at_t(Terms1),e_at_t(Terms2)) as
        no  : no
        yes : eq_terms(e_adv_t(Terms1),e_adv_t(Terms2));
```

A.5.5 Finding Terms and Common Prefixes

The function `same_t` computes a common prefix of two term sequences $\vec{s}$ and $\vec{t}$. If one of the sequences is empty, the common prefix is empty. Otherwise, $\vec{s} = s, \vec{s'}$ and $\vec{t} = t, \vec{t'}$, and the common prefix is empty if $s \neq t$, or s followed by the common prefix of $\vec{s'}$ and $\vec{t'}$ otherwise.

136a ⟨*epic-utl esg* 133a⟩+≡

```
same_t    : Sq_t # Sq_t -> Sq_t {external};
```

136b ⟨*epic-utl rules* 133b⟩+≡

```
same_t(Terms1,Terms2) =
  given e_null(Terms1),e_null(Terms2) as
    yes,YN   : Terms1
    YN, yes  : Terms2
    no, no   : given eq_term(e_at_t(Terms1),e_at_t(Terms2)) as
                 no   : mk_null
                 yes  : mk_indx_t(e_at_t(Terms1),
                                  same_t(e_adv_t(Terms1),e_adv_t(Terms2)));
```

The function `has_t` tests whether a sequence contains a term or not.

136c ⟨*epic-utl esg* 133a⟩+≡

```
has_t : Term # Sq_t -> YN {external};
```

136d ⟨*epic-utl rules* 133b⟩+≡

```
has_t(Term,Sq_t) =
  given e_null(Sq_t) as
    yes : no
    no  : given eq_term(Term,e_at_t(Sq_t)) as
            yes : yes
            no  : has_t(Term,e_adv_t(Sq_t));
```

The function `pos_t` determines an i such that $s = t_i$, under the precondition that $s \in \vec{t}$ (otherwise it loops).

136e ⟨*epic-utl esg* 133a⟩+≡

```
pos_t : Term # Sq_t -> Nat {external};
```

136f ⟨*epic-utl rules* 133b⟩+≡

```
pos_t(Term,Sq_t) =
  given eq_term(Term,e_at_t(Sq_t)) as
    yes : d1
    no  : succ(pos_t(Term,e_adv_t(Sq_t)));
```

A.5.6 Annotating Rules

The functions **cat_id_rules** and **nat_id_rules** annotate the OFS of the LHS of a set of rules, with an `Id` and a `Nat`, respectively.

136g ⟨*epic-utl esg* 133a⟩+≡

```
cat_id_rules  : Id # Sq_r      -> Sq_r {external};
nat_id_rules  : Nat # Sq_r     -> Sq_r {external};
```

For the implementation of **cat_id_rules** and **nat_id_rules**, we need to separate a non-empty sequence of rules in a first rule and the rest, and obtain LHS, RHS and the arguments of the LHS.

137a ⟨*obtain First, Rest, Lhs, Rhs, and Largs from Rules* 137a⟩≡

```
given e_at_r(Rules),e_adv_r(Rules) as First,Rest :
given e_lhs(First),e_rhs(First)    as Lhs,Rhs :
given e_subs_t(Lhs)                as Largs :
```

137b ⟨*epic-utl rules* 133b⟩+≡

```
cat_id_rules(Id,Rules) =
  given e_null(Rules) as
    yes : Rules
    no  : ⟨obtain First, Rest, Lhs, Rhs, and Largs from Rules 137a⟩
          given cat_id(e_ofs(Lhs),Id) as F' :
          given mk_term(F',Largs)     as Lhs' :
            mk_indx_r(mk_rule(Lhs',Rhs),cat_id_rules(Id,Rest));
nat_id_rules(Nat,Rules) =
  given e_null(Rules) as
    yes : Rules
    no  : ⟨obtain First, Rest, Lhs, Rhs, and Largs from Rules 137a⟩
          given nat_id(e_ofs(Lhs),Nat)   as F' :
          given mk_term(F',Largs)        as Lhs' :
            mk_indx_r(mk_rule(Lhs',Rhs),nat_id_rules(Nat,Rest));
```

A.5.7 Making new variables

With **mkvars**, we can construct a number of indexed variables. We use an auxiliary function **mkvars2** to get the indices in the right order.

137c ⟨*epic-utl esg* 133a⟩+≡

```
mkvars   : Str # Nat       -> Sq_t {external};
```

137d ⟨*epic-utl sg* 134f⟩+≡

```
mkvars2 : Nat # Str # Nat -> Sq_t;
```

137e ⟨*epic-utl rules* 133b⟩+≡

```
mkvars(Str,Nat) = mkvars2(d0,Str,Nat);

mkvars2(Nat1,Str,Nat2) =
  given eq_int(Nat1,Nat2) as
    yes : mk_null
    no  : given succ(Nat1) as Nat3 :
            mk_indx_t(mk_var(cat_id(mk_id(Str),mk_id(int2str(Nat3)))),
                      mkvars2(Nat3,Str,Nat2));
```

A.5.8 Finding out about Variable Configurations

The function `nonvar_pos` determines the first nonvariable term in a sequence.

138a ⟨*epic-utl esg* 133a⟩+≡

```
nonvar_pos   : Sq_t      -> Nat {external};
```

138b ⟨*epic-utl rules* 133b⟩+≡

```
nonvar_pos(Sq_t) =
  given e_null(Sq_t) as
    yes : d1
    no  : given e_is_var(e_at_t(Sq_t)) as
            no  : d1
            yes : succ(nonvar_pos(e_adv_t(Sq_t)));
```

The function `all_vars` determines if a sequence consists entirely of variables.

138c ⟨*epic-utl esg* 133a⟩+≡

```
all_vars     : Sq_t      -> YN {external};
```

138d ⟨*epic-utl rules* 133b⟩+≡

```
all_vars(Sq_t) =
  given e_null(Sq_t) as
    yes : yes
    no  : given e_is_var(e_at_t(Sq_t)) as
            no  : no
            yes : all_vars(e_adv_t(Sq_t));
```

Given a number of rules and a number n, `least_nonvar_pos` returns the leftmost position of a nonvariable argument of an LHS when this position is less than n, and n otherwise.

138e ⟨*epic-utl esg* 133a⟩+≡

```
least_nonvar_pos : Sq_r # Nat           -> Nat {external};
```

138f ⟨*epic-utl rules* 133b⟩+≡

```
least_nonvar_pos(Rules,Nat) =
  given e_null(Rules) as
    yes : Nat
    no  : given e_at_r(Rules),e_adv_r(Rules) as Rule,Rest :
          given nonvar_pos(e_subs_t(e_lhs(Rule))) as Nat' :
          given gt(Nat,Nat') as
            yes : least_nonvar_pos(Rest,Nat')
            no  : least_nonvar_pos(Rest,Nat);
```

A.5.9 Selecting Rules

In the expression of the compilation algorithm, selection of rules satisfying certain predicates frequently occurs. We use curried versions of the predicates in order to specify the traversal only once. The predicate `definition` selects rules defining a certain function symbol, the predicate `mg` selects *most general* rules, i.e. rules with an LHS consisting of a single function symbol with only variables as arguments, the predicate `ofs_at` selects rules, whose LHSs have a certain function symbol at a certain argument position, the predicate `leftmost_nonvar` selects the rules for which a certain argument position is the leftmost nonvariable position, and `inv` is used to compute the complement of a selection. Predicates are applied to rules by `apply`, and recursive filtering is done by `select`.

139a ⟨*epic-utl esg* 133a⟩+≡

```
definition      : Id          -> Test {free};
mg              :             -> Test {free};
ofs_at          : Id # Nat    -> Test {free};
leftmost_nonvar : Nat         -> Test {free};
inv             : Test        -> Test {free};
apply           : Test # Rule -> YN {external};
```

139b ⟨*epic-utl rules* 133b⟩+≡

```
apply(inv(Test),Rule) = not(apply(Test,Rule));
apply(definition(F),Rule) = eq_id(F,e_ofs(e_lhs(Rule)));
apply(mg,Rule) = all_vars(e_subs_t(e_lhs(Rule)));
apply(ofs_at(F,N),Rule) =
  given nth_t(N,e_subs_t(e_lhs(Rule))) as Tn :
  given e_is_var(Tn) as
    yes : no
    no  : eq_id(F,e_ofs(Tn));
apply(leftmost_nonvar(N),Rule)
= eq_int(N,nonvar_pos(e_subs_t(e_lhs(Rule))));
```

139c ⟨*epic-utl esg* 133a⟩+≡

```
select          : Test # Sq_r            -> Sq_r {external};
```

139d ⟨*epic-utl rules* 133b⟩+≡

```
    select(Test,Sq_r) =
      given e_null(Sq_r) as
        yes : Sq_r
        no  : given e_at_r(Sq_r),e_adv_r(Sq_r) as First,Rest :
              given apply(Test,First) as
                yes : mk_indx_r(First,select(Test,Rest))
                no  : select(Test,Rest);
```

A.6 An Efficient Representation of MTRS rules

The system delivered by our transformation consists entirely of MTRS rules. In order to achieve a simpler generation of ARM code from the MTRS rules, we use an alternative representation, in which some redundancy is avoided. In order to avoid redundancy in the text below, we will write 'X rule' instead of 'MTRS rule of the form X', where 'X' is one of **R**, **I**, **D**, **A**, **M**, **C**.

For pretty-printing, there is a function `form2e` with turns this representation back into the canonical representation.

140a ⟨*mtrs forms sg* 140a⟩≡

```
    form2e  : Form                                -> Rule;
```

In Sections A.6.1, A.6.2, A.6.3, A.6.4, A.6.5 and A.6.6, we present the various forms. In these sections, we will often construct sequences of variables as follows:

140b ⟨*make VXs* 140b⟩≡

```
  given mkvars(‘X‘,Xs) as VXs :
```

140c ⟨*make VYs* 140c⟩≡

```
  given mkvars(‘Y‘,Ys) as VYs :
```

140d ⟨*make VZs* 140d⟩≡

```
  given mkvars(‘Z‘,Zs) as VZs :
```

A.6.1 Form R, $f(y) \rightarrow y$

An **R** rule is indicated by the following constructor:

140e ⟨*mtrs forms sg* 140a⟩+≡

```
    form_R  : Id                                  -> Form {free};
```

The reconstruction of an **R** rule is given by

140f ⟨*mtrs forms rules* 140f⟩≡

```
    form2e(form_R(F)) =
      given mkvars(‘Y‘,d1)  as VYs :
        mk_rule(mk_term(F,VYs),e_at_t(VYs));
```

A.6.2 Form I, $f(\vec{x}) \rightarrow h(\vec{x})$

An **I** rule is indicated by the following constructor:

140g ⟨*mtrs forms sg* 140a⟩+≡

```
form_I  : Id      # Id # Nat                  -> Form {free};
```

The reconstruction of an **I** rule is given by

141a ⟨*mtrs forms rules* 140f⟩+≡

```
form2e(form_I(F,H,Xs)) =
  ⟨make VXs 140b⟩
  mk_rule(mk_term(F,VXs),mk_term(H,VXs));
```

A.6.3 Form D, $f(\vec{x}, \vec{y}, \vec{z}) \rightarrow h(\vec{x}, \vec{z})$

A **D** rule is indicated by the following constructor:

141b ⟨*mtrs forms sg* 140a⟩+≡

```
form_D  : Id      # Id # Nat # Nat # Nat      -> Form {free};
```

The reconstruction of a **D** rule is given by

141c ⟨*mtrs forms rules* 140f⟩+≡

```
form2e(form_D(F,H,Xs,Ys,Zs)) =
  ⟨make VXs 140b⟩ ⟨make VYs 140c⟩ ⟨make VZs 140d⟩
  given mk_term(F,mk_cat(VXs,mk_cat(VYs,VZs))) as Lhs :
  given mk_term(H,mk_cat(VXs,VZs))             as Rhs :
    mk_rule(Lhs,Rhs);
```

A.6.4 Form A, $f(\vec{x}, \vec{z}) \rightarrow h(\vec{x}, y, \vec{z})$

In an **A** rule, $y = x_i$ or $y = z_i$. The first case is indicated by the constructor `form_AX`, where the last argument is $\mid \vec{x} \mid - i$, and the second case is indicated by the constructor `form_AZ`, where the last argument is i.

141d ⟨*mtrs forms sg* 140a⟩+≡

```
form_AX : Id      # Id # Nat       # Nat # Nat -> Form {free};
form_AZ : Id      # Id # Nat       # Nat # Nat -> Form {free};
```

The reconstruction of an **A** rule is given by

141e ⟨*mtrs forms rules* 140f⟩+≡

```
form2e(form_AX(F,H,Xs,Zs,IX)) =
  ⟨make VXs 140b⟩ ⟨make VZs 140d⟩
  given mk_indx_t(nth_t(IX,rev_t(VXs)),mk_null) as VYs :
  given mk_term(F,mk_cat(VXs,VZs))              as Lhs :
  given mk_term(H,mk_cat(VXs,mk_cat(VYs,VZs)))  as Rhs :
```

```
      mk_rule(Lhs,Rhs);
  form2e(form_AZ(F,H,Xs,Zs,IZ)) =
    ⟨make VXs 140b⟩ ⟨make VZs 140d⟩
    given mk_indx_t(nth_t(IZ,VZs),mk_null)       as VYs :
    given mk_term(F,mk_cat(VXs,VZs))             as Lhs :
    given mk_term(H,mk_cat(VXs,mk_cat(VYs,VZs))) as Rhs :
      mk_rule(Lhs,Rhs);
```

A.6.5 Form C, $f(\vec{x},\vec{y},\vec{z}) \rightarrow h(\vec{x},g(\vec{y}),\vec{z})$

A **C** rule is indicated by the following constructor:

142a ⟨*mtrs forms sg* 140a⟩+≡

```
  form_C  : Id # Id # Id # Nat # Nat # Nat       -> Form {free};
```

The reconstruction of a **C** rule is given by

142b ⟨*mtrs forms rules* 140f⟩+≡

```
  form2e(form_C(F,G,H,Xs,Ys,Zs)) =
    ⟨make VXs 140b⟩ ⟨make VYs 140c⟩ ⟨make VZs 140d⟩
    given mk_term(F,mk_cat(VXs,mk_cat(VYs,VZs)))              as Lhs :
    given mk_term(H,mk_cat(VXs,mk_indx_t(mk_term(G,VYs),VZs))) as Rhs :
      mk_rule(Lhs,Rhs);
```

A.6.6 Form M, $f(\vec{x},g(\vec{y}),\vec{z}) \rightarrow h(\vec{x},\vec{y},\vec{z})$

An **M** rule is indicated by the following constructor:

142c ⟨*mtrs forms sg* 140a⟩+≡

```
  form_M  : Id # Id # Id # Nat # Nat # Nat       -> Form {free};
```

The reconstruction of an **M** rule is given by

142d ⟨*mtrs forms rules* 140f⟩+≡

```
  form2e(form_M(F,G,H,Xs,Ys,Zs)) =
    ⟨make VXs 140b⟩ ⟨make VYs 140c⟩ ⟨make VZs 140d⟩
    given mk_term(F,mk_cat(VXs,mk_indx_t(mk_term(G,VYs),VZs))) as Lhs :
    given mk_term(H,mk_cat(VXs,mk_cat(VYs,VZs)))              as Rhs :
      mk_rule(Lhs,Rhs);
```

A.6.7 Lists of forms

Lists of forms are constructed using `cons` and `nil`. We introduce another list constructor `tcons`, in order to add type information (concerning the types of newly introduced function symbols).

142e ⟨*mtrs forms sg* 140a⟩+≡

```
nil     :                                     -> List_Form {free};
cons    : Form # List_Form                    -> List_Form {free};
tcons   : Type # List_Form                    -> List_Form {free};
append  : List_Form # List_Form               -> List_Form;
```

142f ⟨*mtrs forms rules* 140f⟩+≡

```
append(nil,L) = L;
append(cons(Form,L1),L2) = cons(Form,append(L1,L2));
append(tcons(Type,L1),L2) = tcons(Type,append(L1,L2));
```

With [forms2e]], we can convert *lists* of forms to rules, and with `forms2t`, we can convert a sequence of forms to a sequence of types.

143a ⟨*mtrs forms sg* 140a⟩+≡

```
forms2e : LIST_Form                           -> Sq_r;
forms2t : LIST_Form                           -> Sq_f;
```

143b ⟨*mtrs forms rules* 140f⟩+≡

```
forms2e(nil) = mk_null;
forms2e(cons(Rule,Rules))
= mk_indx_r(form2e(Rule),forms2e(Rules));
forms2e(tcons(Type,Rules))
= forms2e(Rules);

forms2t(nil) = mk_null;
forms2t(cons(Form,Forms)) = forms2t(Forms);
forms2t(tcons(Type,Forms)) = mk_indx_f(Type,forms2t(Forms));
```

A.6.8 The combined module for forms

The representation module for MTRS forms is now given by

143c ⟨*mtrsforms.epg* 143c⟩≡

```
module Mtrsforms
types
⟨yesno esg 128b⟩ ⟨int esg 128c⟩ ⟨char esg 128d⟩ ⟨id esg 130a⟩
⟨epic abstract syntax access 131b⟩ ⟨epic abstract syntax build 132c⟩
⟨epic-utl esg 133a⟩ ⟨mtrs forms sg 140a⟩
rules
⟨mtrs forms rules 140f⟩
```

This code is written to file `mtrsforms.epg`.

A.6.9 The external signature of forms

143d ⟨*mtrs forms esg* 143d⟩≡

```
form_R  : Id                               -> Form      {free};
form_I  : Id      # Id # Nat               -> Form      {free};
form_D  : Id      # Id # Nat # Nat # Nat   -> Form      {free};
form_AX : Id      # Id # Nat       # Nat # Nat -> Form  {free};
form_AZ : Id      # Id # Nat       # Nat # Nat -> Form  {free};
form_C  : Id # Id # Id # Nat # Nat # Nat   -> Form      {free};
form_M  : Id # Id # Id # Nat # Nat # Nat   -> Form      {free};
nil     :                                  -> List_Form {free};
cons    : Form # List_Form                 -> List_Form {free};
tcons   : Type # List_Form                 -> List_Form {free};
append  : List_Form # List_Form            -> List_Form {external};
form2e  : Form                             -> Rule      {external};
forms2e : LIST_Form                        -> Sq_r      {external};
forms2t : LIST_Form                        -> Sq_f      {external};
```

A.7 The translation from TRSs into MTRSs

Now the preliminaries are completed, we turn to the transformation itself. Following Chapter 3, we present the transformation in three stages.

The first stage, found in Section A.7.1, turns any TRS into a *simply complete* TRS. This means that every defined function has a most general rule (i.e., all arguments of the LHS are (linear) variables), and it implies sufficient completeness. This transformation is not correct for arbitrary rewrite relations. In the context of EPIC, it suffices that the transformation is correct for innermost rewriting. See [Ver95] for a study of a broader class of rewrite relations. Fortunately, we do not need such assumptions for the other two transformations.

In the first stage, we also take care that, apart from a single rule of form **R**, RHSs do not consist of a single variable, and that free constructors do not occur as outermost function symbol on a RHS. This is not absolutely necessary, but it simplifies the last transformation.

The second translation, found in Section A.7.2, takes care that every resulting non-MTRS rule is most general. The third translation, found in Section A.7.3, reduces the complexity of the RHS to the level of MTRS rules.

A.7.1 Making the system simply complete

Here we make the system comply with the following requirements:

- Except for external functions, there is at least one most general rule for every function symbol in the original system.
- Free constructors do not occur as outermost function symbol on a RHS.

```
module M
types
  a:   -> S {external};
  b: S -> S {free};
  f: S -> S ;
rules
  f(a)    = a;
  f(b(X)) = X;
```

⇒

```
module M
types
  "#return" : X -> X;
  "b!"      : S -> S {free};
  "f!"      : S -> S {free};
  a         :   -> S {external};
  b         : S -> S ;
  f         : S -> S ;
rules
  "#return"(X) = X;
  b = "#return"("b!");
  f(a) = a;
  f("b!"(X)) = "#return"(X);
  f(X) = "#return"("f!"(X));
```

Figure A.1: An instance of the transformation for simple completeness

- Apart from a single **R** rule, RHSs do not consist of a single variable.

To this end, for every non-external function `f` that is either free, or for which no most general rule is defined, a new function `"f!"` is introduced, the so-called ‘constructor-variant’ of `f`. External functions are left alone by the transformation. Additionally, one unary function `"#return"` is introduced, defined by the rule `"#return"(X)=X`, which is an MTRS rule of form **R**.

Finally, for all constructor variants, a rule of the form

```
f(X1,..,Xn) = "#return"("f!"(X1,..,Xn))
```

is introduced, and all non-root occurrences of function symbols in lhss are replaced by constructor variants. Again, external functions are left alone.

An example of this transformation is given by Figure A.1.

We will now discuss the details of this transformation. First, we find out which (non-external) functions are free constructor, or do not have a most general rule with the function `constructors`

145a ⟨*trscompl sg* 145a⟩≡

```
return_id : -> Id;
```

145b ⟨*trscompl rules* 145b⟩≡

```
return_id = cat_id(mk_id(‘‘),mk_id(‘return‘));
```

Note that we can test if `F` has a most general rule in `Sq_r` as follows:

145c ⟨`Sq_r` *most general for* `F` 145c⟩≡

```
not(e_null(select(mg,select(definition(e_name(F)),Sq_r))))
```

145d ⟨*trscompl sg* 145a⟩+≡

```
constructors : Sq_f # Sq_r -> Sq_f;
```

145e ⟨*trscompl rules* 145b⟩+≡

```
constructors(Sq_f,Sq_r) =
  given e_null(Sq_f) as
    yes : Sq_f
    no  : given e_at_f(Sq_f) as F :
          given e_free(F),e_external(F),e_adv_f(Sq_f) as
            yes,no,Rest : mk_indx_f(F,constructors(Rest,Sq_r))
            no,yes,Rest : constructors(Rest,Sq_r)
            no,no ,Rest :
              given ⟨Sq_r most general for F 145c⟩ as
                yes : constructors(Rest,Sq_r)
                no  : mk_indx_f(F,constructors(Rest,Sq_r));
```

Then we produce construction forms for these functions, given the `Id` of the `return` function. For every new function symbol, a corresponding type is introduced. Here, the type is derived straightforwardly from the type (argument sorts and result sorts) of the original function. It should be noted that in EPIC, only the number of arguments is relevant, and all other typing information should be regarded as documentation only.

Nevertheless, it would be better to provide the way in which types are derived from other types as a parameter. We will not pursue this matter here, and use type derivations that are correct at least for first-order many-sorted rewriting systems without overloading.

The function `crules` produces the rules that transform functions into their constructor variants.

146a ⟨*trscompl sg* 145a⟩+≡

```
crules   : Sq_f # Id  -> List_Form;
```

146b ⟨*trscompl rules* 145b⟩+≡

```
crules(Sq_f,Rid) =
  given e_null(Sq_f) as
    yes : nil
    no  : given e_at_f(Sq_f),e_adv_f(Sq_f) as Fid,Rest :
          given cvar(e_name(Fid)) as F_c :
            tcons(mk_type(F_c,e_asorts(Fid),e_rsort(Fid)),
             cons(form_C(e_name(Fid),F_c,Rid,d0,e_arity(Fid),d0),
                  crules(Rest,Rid)));
```

For the LHSs, we need to replace all non-outermost function symbols by constructor variants. To this end we use `cvar_term`:

146c ⟨*trscompl sg* 145a⟩+≡

```
cvar_term   : Term -> Term;
cvar_terms  : Sq_t -> Sq_t;
```

146d ⟨*trscompl rules* 145b⟩+≡

```
cvar_term(Term) =
  given e_is_var(Term) as
    yes : Term
    no  : mk_term(cvar(e_ofs(Term)),cvar_terms(e_subs_t(Term)));
cvar_terms(Sq_t) =
  given e_null(Sq_t) as
    yes : Sq_t
    no  : mk_indx_t(cvar_term(e_at_t(Sq_t)),cvar_terms(e_adv_t(Sq_t)));
```

For the RHS, we add a `return` function when the RHS consists of a single variable. The combined transformation of LHS and RHS is done by `fix_rules`.

147a ⟨*trscompl sg* 145a⟩+≡

```
fix_rules  : Sq_r # Id -> Sq_r;
```

147b ⟨*trscompl rules* 145b⟩+≡

```
fix_rules(Sq_r,Rid) =
  given e_null(Sq_r) as
    yes : Sq_r
    no  : given e_at_r(Sq_r) as Rule :
          given e_lhs(Rule),e_rhs(Rule) as Lhs,Rhs :
          given e_ofs(Lhs),cvar_terms(e_subs_t(Lhs)) as F,Largs' :
          given e_is_var(Rhs),mk_term(F,Largs'),e_adv_r(Sq_r) as
            no ,Lhs',Rest :
                mk_indx_r(mk_rule(Lhs',Rhs),fix_rules(Rest,Rid))
            yes,Lhs',Rest :
                given mk_term(Rid,mk_indx_t(Rhs,mk_null)) as Rhs' :
                  mk_indx_r(mk_rule(Lhs',Rhs'),fix_rules(Rest,Rid));
```

The entire simple completeness transformation is implemented by `complete`

147c ⟨*trscompl sg* 145a⟩+≡

```
complete : Mod -> Mod;
```

147d ⟨*trscompl rules* 145b⟩+≡

```
complete(Mod) =
  given e_subs_f(Mod), e_subs_r(Mod)                      as Types,Rules :
  given constructors(Types,Rules),return_id               as Constrs,Rid :
  given crules(Constrs,Rid)                               as Cforms :
  given forms2e(Cforms),forms2t(Cforms)                   as Crules,Ctypes :
  given mk_var(mk_id(`X`))                                as Sort       :
  given mk_type(return_id,mk_indx_t(Sort,mk_null),Sort)  as ReturnType :
  given fix_rules(Rules,Rid),mk_indx_f(ReturnType,Types) as Rules',Types' :
    mk_mod(mk_cat(Types',Ctypes),mk_cat(Rules',Crules));
```

147e

```
⟨trscompl.epg 147e⟩≡
  module Trscompl
  types
  ⟨yesno esg 128b⟩ ⟨char esg 128d⟩ ⟨int esg 128c⟩ ⟨id esg 130a⟩
  ⟨epic abstract syntax access 131b⟩ ⟨epic abstract syntax build 132c⟩
  ⟨epic-utl esg 133a⟩ ⟨mtrs forms esg 143d⟩
  ⟨trscompl sg 145a⟩
  rules
  ⟨trscompl rules 145b⟩
```

This code is written to file `trscompl.epg`.

A.7.2 The LHS transformation

The function `t2m_lhs` performs the LHS transformation on a sequence of rules defining a single function symbol f. The transformation for multiple functions is the union of the transformed systems for the individual symbols. The second argument of `t2m_lhs` contains the types of all functions occurring in the EPIC program. This information is used to compute types for the functions that are introduced by the transformation.

148

```
⟨lhs2mtrs sg 148⟩≡
    t2m_lhs           : Sq_r # Sq_f       -> List_Form {external};
```

When all rules defining f are most general, no matching occurs, and the transformation for the RHS construction is called on the first rule. We can ignore the other rules here, because the semantics of EPIC just specifies that some alternative should be taken. In order to use the compiler in this chapter for the study of TRSs in general, all alternatives should be transformed. However, using the current naming scheme, this would lead to name-clashes. At the expense of longer names, this problem can be solved.

For the case that not all rules are most general, we quote the transformation described in Chapter 3:

> Let i be the least index such that for some rule $f(\vec{t}) \to s \in R$, $t_i \notin V$, and let $G = \{g | f(\vec{t}) \to r \in R \wedge t_i = g(\vec{u})\}$, the set of function symbols found at position i of LHSs defining f in R. Then, taking $|\ \vec{x}\ | = |\ \vec{t}\ | = i - 1$, and $f^S, f_g \notin \Sigma$ fresh symbols:
>
> $$[\text{sim-rec}] \ \sim \langle \Sigma, R \rangle \ = \ (\bigcup_{g \in G} \mathit{Match}_g \cup \mathit{Matched}_g) \cup \mathit{Skip} \cup \mathit{Other},$$

where

$$\begin{array}{rcl}
\mathit{Match}_g & = & \langle\{f, g, f_g\}, \{m1 : f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z})\}\rangle; \\
\mathit{Matched}_g & = & \sim\langle\Sigma \cup \{f_g, f^S\},\ \{m2 : f_g(\vec{t}, \vec{u}, \vec{v}) \to r | m3 : f(\vec{t}, g(\vec{u}), \vec{v}) \to r \in R\} \\
 & & \qquad \cup\{m4 : f_g(\vec{x}, \vec{y}, \vec{z}) \to f^S(\vec{x}, g(\vec{y}), \vec{z})\}\rangle; \\
\mathit{Skip} & = & \langle\Sigma \cup \{f^S\}, \{s1 : f(\vec{x}) \to f^S(\vec{x})\}\rangle; \\
\mathit{Other} & = & \sim\langle\Sigma \cup f^S,\ \{o1 : f^S(\vec{t}) \to s | o2 : f(\vec{t}) \to s \in R \wedge\ \vec{t}_i \in V\} \\
 & & \qquad \cup\{o3 : r | r \in R \wedge \mathit{ofs}(\mathit{lhs}(r)) \neq f\}\rangle.
\end{array}$$

An intuitive explanation of [sim-rec] is, that Match_g has a rule $m1$ that matches a symbol g at position i, $\mathit{Matched}_g$ deals with a succesful match of g at position i, by either completing a match of $m3$ by applying $m2$, or restoring the LHS of $m1$ (up to f^S) by applying $m4$, *Skip* just replaces f by f^S, and *Other* simulates the rules $o2$ that have a variable at position i through rules $o1$, and rules $o3$ for other function symbols than f.

This is an almost accurate description of algorithm below, except that `t2m_lhs` is only called on rules defining a single function symbol, so the rules $o3$ are irrelevant, and we take into account the number of terms on the T stack, so an **I** rule is added when needed.

We first compute the first rule `Rule`, its LHS `Lhs`, OFS `F`, arity `Arity`, and the least nonvariable position `L` from a sequence of rules:

149a ⟨*get* `Rule`, `Lhs`, `F`, `Arity` *and* `L` *from* `Rules` 149a⟩≡

```
given e_at_r(Rules)                      as Rule  :
given e_lhs(Rule)                        as Lhs   :
given e_ofs(Lhs)                         as F     :
given e_number(e_subs_t(Lhs))            as Arity :
given least_nonvar_pos(Rules,succ(Arity)) as L     :
```

In order to execute a match on position `L`, there should be exactly `min(L,d1)` terms on the T stack. To account for the situation that `F` does not have the right number of terms on the `T` stack, a function `F'` is computed which *does* have the right configuration, and an **I** rule `Skip` to replace `F` by `F'`.

149b ⟨*get* `NewAnn,S,Type_F',Types',Skip` *from* `F,Arity` 149b⟩≡

```
given min(L,d1)                                    as NewAnn :
given mk_id(int2str(NewAnn))                       as S      :
given cat_id(nat_id(F,NewAnn),S)                   as F' :
given type_of(F,Types)                             as Type_F :
given mk_type(F',e_asorts(Type_F),e_rsort(Type_F)) as Type_F'  :
given mk_indx_f(Type_F',Types)                     as Types'   :
given form_I(F,F',Arity)                           as Skip :
```

For all rules with a non-variable subterm at position `L`, rules simulating a match at this position, as well as rules restoring the term at this position are generated by the function `match_restore` (see below).

The function `least_nonvar_pos` is used to find the leftmost position `L` on which some rule has a non-variable subterm. Then, `leftmost_nonvar(L)` is used to separate the rules that have a variable at this position from the rules that have a non-variable sub-term at this position.

149c ⟨*make* `MatchRestore` *from* `L,Rules,Types'` 149c⟩≡

```
  given match_restore(L,select(leftmost_nonvar(L),Rules),Types')
     as MatchRestore :
```

For the other rules, `t2m_lhs` is invoked recursively as follows

149d ⟨*make* `Other` *from* `S,L,Rules,Types'` 149d⟩≡

```
  given t2m_lhs(cat_id_rules(S,select(inv(leftmost_nonvar(L)),
                              Rules)),Types')                  as Other :
```

Combining the bits and pieces above, the rule defining the LHS transformation reads as follows:

149e ⟨*lhs2mtrs rules* 149e⟩≡

```
    t2m_lhs(Rules,Types) =
      given e_null(Rules) as
        yes : mk_null
        no  :
          ⟨get Rule, Lhs, F, Arity and L from Rules 149a⟩
          given eq_int(L,succ(Arity)) as
            yes : t2m_rhs(Rule,Types)
            no  :
              ⟨get NewAnn,S,Type_F',Types',Skip from F,Arity 149b⟩
              given eq_int(NewAnn,get_nat(F)) as
                no  : tcons(Type_F',cons(Skip,
                        t2m_lhs(cat_id_rules(S,nat_id_rules(NewAnn,Rules)),
                                Types')))
                yes : ⟨make MatchRestore from L,Rules,Types' 149c⟩
                      ⟨make Other from S,L,Rules,Types' 149d⟩
                      tcons(Type_F',cons(Skip,
                        append(MatchRestore,Other)));
```

The rules for matching at position `N`, and restoring the term in case of subsequent failure, are constructed by the function `match_restore`. If `Rules` is nonempty, the outermost function symbol `G` of the `N`th argument of the LHS of the first rule is taken. The rule `Match_g` is an M rule to match `G` at this position, and `Rest_g` is a rule to restore the term headed by `G` in case of subsequent failure. The function `flatten_rules` transforms all rules that have `G` at position `N` into rules where this symbol is removed, and recursively, `match_restore` is applied to the rules that have another function symbol than `G` at `N`.

150a ⟨*lhs2mtrs sg* 148⟩+≡

```
    match_restore       : Nat # Sq_r # Sq_f -> List_Form;
    restore_when_needed : Rule # Sq_r       -> Sq_r;
```

150b ⟨*lhs2mtrs rules* 149e⟩+≡

```
match_restore(N,Rules,Types) =
  given e_null(Rules) as
    yes : Rules
    no  :
      given  e_lhs(e_at_r(Rules))                         as Lhs     :
      given  e_ofs(Lhs)                                   as F       :
      given  type_of(F,Types)                             as Type_F  :
      given  e_subs_t(Lhs)                                as Largs   :
      given  e_number(Largs)                              as Arity   :
      given  e_ofs(nth_t(N,Largs))                        as G       :
      given  type_of(G,Types)                             as Type_G  :
      given  cat_id(F,G)                                  as H       :
      given  pre_n_t(N,e_asorts(Type_F))                  as PreN    :
      given  adv_n_t(N,e_asorts(Type_F))                  as PostN   :
      given  e_asorts(Type_G)                             as Gargs   :
      given  mk_cat(Gargs,PostN)                          as FromN   :
      given  e_rsort(Type_F)                              as Rsort   :
      given  mk_type(H,mk_cat(PreN,FromN),Rsort)          as Type_H  :
      given  mk_id(int2str(min(N,d1)))                    as S       :
      given  cat_id(F,S)                                  as F'      :
      given  mk_type(F',e_asorts(Type_F),Rsort)           as Type_F' :
      given  mk_indx_f(Type_F',mk_indx_f(Type_H,Types))   as Types'  :
      given  min(N,d1)                                    as NX      :
      given  e_number(e_subs_t(nth_t(N,Largs)))           as NY      :
      given  min(Arity,N)                                 as NZ      :
      given  flatten_rules(N,select(ofs_at(G,N),Rules),H) as Frules  :
      given  form_M(F,G,H,NX,NY,NZ)                       as Match_g :
      given  form2e(form_C(H,G,F',NX,NY,NZ))              as Rest_g  :
      given  select(inv(ofs_at(G,N)),Rules)               as NotG    :
      given  match_restore(N,NotG,Types')                 as Oforms  :
      given  restore_when_needed(Rest_g,Frules)           as Rest    :
        append(Oforms,
              tcons(Type_H,
              tcons(Type_F',
              cons(Match_g,t2m_lhs(Rest,Types')))));
```

151a ⟨*lhs2mtrs rules* 149e⟩+≡

```
restore_when_needed(Restore,Frules) =
  given e_null(select(mg,Frules)) as
    no  : Frules
    yes : mk_cat(Frules,mk_indx_r(Restore,mk_null));
```

The function `flatten_rules(N,R,H)` is used to make variants of the rules in `R`, replacing the `N`th argument of their Lhs's by its arguments, and replacing the outermost function symbol by `H`.

151b ⟨*lhs2mtrs sg* 148⟩+≡

```
flatten_rules : NAT # Sq_r # SYM -> Sq_r;
flatten_rule  : NAT # RULE # SYM -> RULE;
```

151c ⟨*lhs2mtrs rules* 149e⟩+≡

```
flatten_rules(N,Rules,H) =
  given e_null(Rules) as
    yes : Rules
    no  : mk_indx_r(flatten_rule(N,e_at_r(Rules),H),
                    flatten_rules(N,e_adv_r(Rules),H));
flatten_rule(N,Rule,H) =
  given e_subs_t(e_lhs(Rule))                           as Largs    :
  given pre_n_t(N,Largs)                                as Before_N :
  given e_subs_t(nth_t(N,Largs))                        as Below_N  :
  given adv_n_t(N,Largs)                                as After_N  :
  given e_rhs(Rule)                                     as Rhs      :
  given mk_term(H,mk_cat(Before_N,mk_cat(Below_N,After_N))) as Lhs  :
    mk_rule(Lhs,Rhs);
```

The entire module defining the LHS transformation is given by the following chunk:

152 ⟨*lhs2mtrs.epg* 152⟩≡

```
module Lhs2mtrs
types
  ⟨yesno esg 128b⟩ ⟨int esg 128c⟩ ⟨char esg 128d⟩ ⟨id esg 130a⟩
  ⟨epic abstract syntax access 131b⟩ ⟨epic abstract syntax build 132c⟩
  ⟨epic-utl esg 133a⟩ ⟨mtrs forms esg 143d⟩
  t2m_rhs: Rule # Types -> X {external};
  ⟨lhs2mtrs sg 148⟩
rules
  ⟨lhs2mtrs rules 149e⟩
```

This code is written to file `lhs2mtrs.epg`.

As an example of the transformation presented in this section, consider Figure A.2. The input program has nested patterns (`g` in the first rule), and `f` has a most general rule. In order to save space, we only show the rules of the transformed system, not the signature. To keep things simple, we have chosen all RHSs identical.

A.7.3 Translating the RHS

We now turn to the RHS transformation.

The function `t2m_rhs` applies the RHS transformation to a single most general rule defining a function symbol f. This is sufficient for the use of EPIC as a programming language, where an arbitrary choice is made if the TRS is ambiguous. Simply applying the transformation given here to *all* rules defining f is not correct, because the names of new

```
                                   rules
                                     f(X_1,X_2) = "f#0"(X_1,X_2);
module M                             f(g(Y_1),Z_1) = "f#g"(Y_1,Z_1);
types                                "f#g"(X_1,X_2) = "f#g#0"(X_1,X_2);
  f: F1 # F2 -> F3;                  "f#g"(g(Y_1),Z_1) = "f#g#g"(Y_1,Z_1);
  g: G1      -> G2;                  "f#g#g"(X_1,X_2) = "f#g#g#1"(X_1,X_2);
  h: H1      -> H2;          ⇒       "f#g#g#1"(X_1,X_2) = "f#g#g#1#1"(X_1,X_2);
rules                                "f#g#g#1"(X_1,h(Y_1)) = "f#g#g#1#h"(X_1,Y_1);
  f(g(g(X)),h(Y)) = f(X,Y);          "f#g#g#1#h"(X_1,X_2) = f(X_1,X_2);
  f(g(X),Y) = f(X,Y);                "f#g#g#1#1"(X_1,X_2) = "f#g#g#1#1#0"(X_1,X_2);
  f(X,Y) = f(X,Y);                   "f#g#g#1#1#0"(Y_1,Z_1) = "f#g#0"(g(Y_1),Z_1);
                                     "f#g#0"(X_1,X_2) = f(X_1,X_2);
                                     "f#0"(X_1,X_2) = f(X_1,X_2);
```

Figure A.2: An instance of the LHS transformation

functions are a function of f, not of the rule under consideration. At the expense of still longer names, this could be fixed.

153a ⟨*rhs2mtrs sg* 153a⟩≡

```
t2m_rhs        : Rule # Types        -> LIST_Form;
```

Given the simple completeness transformation of Section A.7.1, any remaining non-MTRS rule with a most general LHS can be brought into the form

$$f(\vec{x},\vec{y},\vec{z}) \to h(\vec{x},\vec{t},\vec{z})$$

where first $\mid \vec{x} \mid$ is maximized, and then $\mid \vec{z} \mid$. Then, depending on the form of $\vec{t}$, the length of $\vec{y}$, and $L(f)$, we can distinguish several cases. The amount of similarity between $\vec{t}$ and $\vec{y}$ is determined by the function `ts_type`.

153b ⟨*rhs2mtrs sg* 153a⟩+≡

```
ts_type   : Sq_t # Sq_t        -> RHS_TYPE;
ts_empty  :                    -> RHS_TYPE {free};
ts_var    : Term # Sq_t        -> RHS_TYPE {free};
ts_nonvar : Term # Sq_t # Sq_t -> RHS_TYPE {free};
ts_con    : Id                 -> RHS_TYPE {free};
```

153c ⟨*rhs2mtrs rules* 153c⟩≡

```
ts_type(Ts,Ys) =
  given e_null(Ts) as
    yes : ts_empty
    no  : given e_at_t(Ts) as T :
          given e_is_var(T),e_subs_t(T),e_adv_t(Ts) as
```

```
               yes,Args,Ts' : ts_var(T,Ts')
               no ,Args,Ts' : given eq_terms(Ys,Args),e_ofs(T) as
                                yes,G : ts_con(G)
                                no ,G : ts_nonvar(G,Args,Ts');
```

After computing the relevant parameters, `t2m_rhs` calls `t2m_rhs_table`, which has one rule for every case.

153d ⟨*rhs2mtrs sg* 153a⟩+≡

```
t2m_rhs_table : RHS_TYPE # YN # YN #
                Id # Id # Nat # Nat # Nat #
                Sq_t # Sq_t # Sq_t # Sq_f  -> List_Form;
```

154a ⟨*rhs2mtrs rules* 153c⟩+≡

```
t2m_rhs(Rule,Types) =
  given e_lhs(Rule),e_rhs(Rule)            as Lhs,Rhs :
  given e_ofs(Lhs),e_ofs(Rhs)              as F,H :
  given e_subs_t(Lhs),e_subs_t(Rhs)        as Largs,Rargs :
  given same_t(Largs,Rargs)                as Xs :
  given e_number(Xs)                       as NX :
  given adv_n_t(NX,Largs),adv_n_t(NX,Rargs) as Largs2,Rargs2 :
  given rev_t(Largs2),rev_t(Rargs2)        as RLargs2,RRargs2 :
  given rev_t(same_t(RLargs2,RRargs2))     as Zs :
  given e_number(Zs)                       as NZ :
  given min(e_number(Largs2),NZ)           as NY :
  given pre_n_t(plus(NY,d1),Largs2)        as Ys :
  given rev_t(adv_n_t(NZ,RRargs2))         as Ts :
    t2m_rhs_table(ts_type(Ts,Ys),eq_int(NY,d0),eq_int(NX,get_nat(F)),
                  F,H,NX,NY,NZ,Xs,Ys,Zs,Types);
```

The case that $|\vec{y}| = |\vec{t}| = 0$

This is already an **I** rule

$$f(\vec{x}) \to h(\vec{x}),$$

because $|\vec{y}| = 0$ and $|\vec{t}| = 0$ imply that $|\vec{z}| = 0$. For **I** rules, there are no restrictions on $L(f)$ and $L(h)$, so the following case for `t2m_rhs_table` suffices:

154b ⟨*rhs2mtrs rules* 153c⟩+≡

```
t2m_rhs_table(ts_empty,yes,YN1,F,H,NX,NY,NZ,Xs,Ys,Zs,Types)
= cons(form_I(F,H,plus(NX,NZ)),nil);
```

The case that $\mid \vec{y} \mid \neq 0$, but $\mid \vec{t} \mid = 0$ and $L(f) = \mid \vec{x} \mid$

This is the rule

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow h(\vec{x}, \vec{z})$$

where it may be the case that $L(h) \neq L(f)$. This rule is simulated by a **D** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow f^{R0}(\vec{x}, \vec{z})$$

followed by an **I** rule covered by Section A.7.3:

$$f^{R0}(\vec{x}, \vec{z}) \rightarrow h(\vec{x}, \vec{z}).$$

So, we write the following case for `t2m_rhs_table`:

154c ⟨*rhs2mtrs rules* 153c⟩+≡

```
t2m_rhs_table(ts_empty,no,yes,F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
  given cat_id(F,mk_id(int2str(NX)))                          as F'      :
  given type_of(F,Types)                                      as Type_F  :
  given e_asorts(Type_F)                                      as Asorts  :
  given pre_n_t(plus(NX,d1),Asorts)                           as Xsorts  :
  given adv_n_t(plus(NX,NY),Asorts)                           as Zsorts  :
  given mk_type(F',mk_cat(Xsorts,Zsorts),e_rsort(Type_F)) as Type_F' :
    tcons(Type_F',
   cons(form_D(F,F',NX,NY,NZ),
   cons(form_I(F',H,plus(NX,NZ)),
        nil)));
```

The case that $\mid \vec{y} \mid \neq 0$ and $L(f) \neq \mid \vec{x} \mid$, but $\mid \vec{t} \mid = 0$

This looks like a **D** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow h(\vec{x}, \vec{z}),$$

but $L(f)$ is wrong. This can be simulated an **I** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow f^{RA}(\vec{x}, \vec{y}, \vec{z})$$

with $L(f^{RA}) = \mid \vec{x} \mid$, and a rule covered by the case of Section A.7.3:

$$f^{RA}(\vec{x}, \vec{y}, \vec{z}) \rightarrow h(\vec{x}, \vec{z}).$$

This explains the following case for `t2m_rhs_table`:

155a ⟨*rhs2mtrs rules* 153c⟩+≡

```
t2m_rhs_table(ts_empty,no,no,F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
  given type_of(F,Types)                           as Type_F  :
  given cat_id(nat_id(F,NX),mk_id(int2str(NX)))    as F'      :
```

```
    given mk_type(F',e_asorts(Type_F),e_rsort(Type_F)) as Type_F' :
    given mk_indx_t(Type_F',Types)                     as Types'  :
      tcons(Type_F',
       cons(form_I(F,F',plus(NX,plus(NY,NZ))),
            t2m_rhs_table(ts_empty,no,yes,F',H,
                          NX,NY,NZ,Xs,Ys,Zs,Types')));
```

The case that $\vec{t} = g(\vec{y})$ and $L(f) = \mid \vec{x} \mid$

This is already a **C** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z}),$$

therefore we have the following case for `t2m_rhs_table`:

155b ⟨*rhs2mtrs rules* 153c⟩+≡

```
  t2m_rhs_table(ts_con(G),YN1,yes,F,H,NX,NY,NZ,Xs,Ys,Zs,Types)
  = cons(form_C(F,G,H,NX,NY,NZ),nil);
```

The case that $\vec{t} = g(\vec{y})$ and $L(f) \neq \mid \vec{x} \mid$

Apart from the $L(f)$, this is a **C** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z}).$$

With an **I** rule, this case can be reduced to the one in Section A.7.3:

156a ⟨*rhs2mtrs rules* 153c⟩+≡

```
  t2m_rhs_table(ts_con(G),YN1,no,F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
    given type_of(F,Types)                             as Type_F  :
    given cat_id(nat_id(F,NX),mk_id(int2str(NX)))      as F'      :
    given mk_type(F',e_asorts(Type_F),e_rsort(Type_F)) as Type_F' :
    given mk_indx_t(Type_F',Types)                     as Types'  :
     tcons(Type_F',
     cons(form_I(F,F',plus(NX,plus(NY,NZ))),
         t2m_rhs_table(ts_con(G),YN1,yes,F',H,
                       NX,NY,NZ,Xs,Ys,Zs,Types')));
```

The case that $\vec{t} = v, \vec{t'}$, and $L(f) = \mid \vec{x} \mid$

This is the rule

$$f(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, v, \vec{t'}, \vec{z})$$

To simulate this, we need an **A** rule

$$f(\vec{x}, \vec{y}, \vec{z}) \to f'(\vec{x}, v, \vec{y}, \vec{z})$$

and a rule which has $|\vec{t}|$ one less (take $\vec{x'}=\vec{x},v$)

$$f'(\vec{x'},\vec{y},\vec{z}) \to h(\vec{x'},\vec{t'},\vec{z})$$

The function `make_A` makes a form with `form_AX` or `form_AZ` depending on the origin of the new variable v.

156b ⟨*rhs2mtrs sg* 153a⟩+≡

```
make_A : YN # Sq_t # YN # Sq_t # Term # Id # Id -> Form;
```

156c ⟨*rhs2mtrs rules* 153c⟩+≡

```
make_A(yes,RXs,no,YsZs,Var,F,FR)
= form_AX(F,FR,e_number(RXs),e_number(YsZs),pos_t(Var,RXs));
make_A(no,RXs,yes,YsZs,Var,F,FR)
= form_AZ(F,FR,e_number(RXs),e_number(YsZs),pos_t(Var,YsZs));
```

Using `make_A`, the **A** rule is constructed below, and `t2m_rhs_table` is called recursively to generate MTRS rules from the simplified rule.

157a ⟨*rhs2mtrs rules* 153c⟩+≡

```
  t2m_rhs_table(ts_var(Var,Ts'),YN1,yes,F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
  given cat_id(F,mk_id('RB')),get_nat(F)            as FR,Ann         :
  given rev_t(Xs),mk_cat(Ys,Zs)                     as RXs,YsZs       :
  given make_A(has_t(Var,RXs),RXs,
               has_t(Var,YsZs),YsZs,Var,F,FR)       as FormA          :
  given type_of(F,Types),type_of(H,Types)           as Type_F,Type_H  :
  given e_at_t(adv_n_t(NX,e_asorts(Type_H)))        as NewSort        :
  given pre_n_t(plus(NX,d1),e_asorts(Type_F))       as Xsorts         :
  given adv_n_t(NX,e_asorts(Type_F))                as YZsorts        :
  given mk_cat(Xsorts,mk_indx_t(NewSort,YZsorts))   as XYZsorts       :
  given mk_type(FR,XYZsorts,e_rsort(Type_F))        as Type_FR        :
  given mk_indx_f(Type_FR,Types)                    as Types'         :
    tcons(Type_FR,
     cons(FormA,
          t2m_rhs_table(ts_type(Ts',Ys),YN1,eq_int(Ann,succ(NX)),
                        FR,H,succ(NX),NY,NZ,rev_t(mk_indx_t(Var,RXs)),
                        Ys,Zs,Types')));
```

The case that $\vec{t}=v,\vec{t'}$, but $L(f) \neq |\vec{x}|$

With an **I** rule

$$f(\vec{x},\vec{y},\vec{z}) \to f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}),$$

this can be reduced to the case above:

$$f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) \to h(\vec{x},v,\vec{t'},\vec{z}).$$

157b ⟨*rhs2mtrs rules* 153c⟩+≡

```
  t2m_rhs_table(ts_var(Var,Ts'),YN1,no,F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
    given cat_id(nat_id(F,NX),mk_id(int2str(NX)))        as FR      :
    given mk_type(FR,e_asorts(type_of(F,Types)),
                     e_rsort(type_of(F,Types)))           as Type_FR :
    given mk_indx_f(Type_FR,Types)                        as Types'  :
      tcons(Type_FR,
      cons(form_I(F,FR,plus(NX,plus(NY,NZ))),
           t2m_rhs_table(ts_var(Var,Ts'),YN1,yes,
                         FR,H,NX,NY,NZ,Xs,Ys,Zs,Types')));
```

The case that $\vec{t} = g(\vec{u}), \vec{t'}$

This is a rule

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow h(\vec{x}, g(\vec{u}), \vec{t'}, \vec{z}),$$

which is simulated by one skip rule to avoid a name conflict (all new function symbols are derived from f)

$$f(\vec{x}, \vec{y}, \vec{z}) \rightarrow f^N(\vec{x}, \vec{y}, \vec{z})$$

a rule that has one function symbol less in $\vec{t}$:

$$f^N(\vec{x}, \vec{y}, \vec{z}) \rightarrow h^R(\vec{x}, \vec{u}, \vec{t'}, \vec{z}),$$

a **C** rule constructing $g(\vec{u})$

$$h^R(\vec{x}, \vec{y'}, \vec{z'}) \rightarrow h^{RR}(\vec{x}, g(\vec{y'}), \vec{z'}),$$

and an **I** rule

$$h^{RR}(\vec{x'}) \rightarrow h(\vec{x'}),$$

which is defined by the rule below:

158 ⟨*rhs2mtrs rules* 153c⟩+≡

```
  t2m_rhs_table(ts_nonvar(G,Targs,Ts'),YN1,YN2,
                F,H,NX,NY,NZ,Xs,Ys,Zs,Types) =
    given cat_id(F,mk_id(`N')),type_of(F,Types)        as FN,Type_F :
    given mk_type(FN,e_asorts(Type_F),e_rsort(Type_F))  as Type_FN :
    given e_number(Targs),plus(NZ,e_number(Ts'))        as NY',NZ' :
    given nat_id(cat_id(F,mk_id(int2str(NX))),NX)       as HR :
    given cat_id(HR,mk_id(`R'))                         as HRR :
    given type_of(H,Types),type_of(G,Types)             as Type_H,Type_G :
    given mk_type(HRR,e_asorts(Type_H),e_rsort(Type_H)) as Type_HRR :
    given pre_n_t(plus(NX,d1),e_asorts(Type_H))         as Xsorts :
    given adv_n_t(plus(NX,d1),e_asorts(Type_H))         as Zsorts :
```

```
given e_asorts(Type_G)                              as Ysorts :
given mk_cat(Xsorts,mk_cat(Ysorts,Zsorts))          as XYZsorts :
given mk_type(HR,XYZsorts,e_rsort(Type_H))          as Type_HR :
given mk_indx_f(Type_FN,
      mk_indx_f(Type_HR,
      mk_indx_f(Type_HRR,Types)))                   as Types' :
given mk_term(FN,mk_cat(Xs,mk_cat(Ys,Zs)))          as Lhs' :
given mk_cat(Xs,mk_cat(Targs,mk_cat(Ts',Zs)))       as Rargs' :
given mk_term(HR,Rargs')                            as Rhs' :
  tcons(Type_FN,
  tcons(Type_HR,
  tcons(Type_HRR,
  cons(form_I(F,FN,plus(NX,plus(NY,NZ))),
  append(t2m_rhs(mk_rule(Lhs',Rhs'),Types'),
         cons(form_C(HR,G,HRR,NX,NY',NZ'),
         cons(form_I(HRR,H,plus(plus(NX,d1),NZ')),
         nil)))))));
```

The whole of Rhs2Mtrs

159a ⟨*rhs2mtrs.epg* 159a⟩≡

```
module Rhs2Mtrs
types
⟨yesno esg 128b⟩ ⟨int esg 128c⟩ ⟨char esg 128d⟩ ⟨id esg 130a⟩
⟨epic abstract syntax access 131b⟩ ⟨epic abstract syntax build 132c⟩
⟨mtrs forms esg 143d⟩ ⟨epic-utl esg 133a⟩ ⟨rhs2mtrs sg 153a⟩
rules
⟨rhs2mtrs rules 153c⟩
```

This code is written to file `rhs2mtrs.epg`.

As an example of an input to the RHS transformation, see Figure A.3. In Figure A.4, the rules resulting from the RHS transformation on the TRS in Figure A.3 are shown.

```
module M
types
  f: S # S # S -> S ;
  h: S # S # S -> S ;
  g: S -> S ;
rules
  f(X,Y,Z) = h(Y,g(X),Z);
```

Figure A.3: Input for the RHS transformation

```
rules
  f(Z_1,Z_2,Z_3) = "f#RB"(Z_2,Z_1,Z_2,Z_3);
  "f#RB"(X_1,X_2,X_3,X_4) = "f#RB#N"(X_1,X_2,X_3,X_4);
  "f#RB#N"(X_1,X_2,X_3,X_4) = "f#RB#N#2"(X_1,X_2,X_3,X_4);
  "f#RB#N#2"(X_1,X_2,Y_1,Z_1) = "f#RB#N#2#2"(X_1,X_2,Z_1);
  "f#RB#N#2#2"(X_1,X_2,X_3) = "f#RB#1"(X_1,X_2,X_3);
  "f#RB#1"(X_1,Y_1,Z_1) = "f#RB#1#R"(X_1,g(Y_1),Z_1);
  "f#RB#1#R"(X_1,X_2,X_3) = h(X_1,X_2,X_3);
```

Figure A.4: Output of the RHS transformation

A.7.4 Tying all Phases Together

159b ⟨*trs2mtrs sg* 159b⟩≡

```
t2m          :      Sq_f # Sq_r # Sq_f -> List_Form;
```

160a ⟨*trs2mtrs rules* 160a⟩≡

```
t2m(Signs,Rules,Types) =
  given e_null(Signs) as
    yes : nil
    no  : append(t2m_lhs(select(definition(e_name(e_at_f(Signs))),Rules),
                         Types),
          t2m(e_adv_f(Signs),
              select(inv(definition(e_name(e_at_f(Signs)))),Rules),Types));
```

We use an externally defined function `p_epic_spec` to parse EPIC programs, and the function `pp_epic_spec` to print EPIC programs. The top-level function, which parses a program, transforms it to an MTRS, and prints this as a program, is called `ppp_trs2mtrs` (mnemonic for *p*arse and pretty-*p*rint from *trs* to *mtrs*).

160b ⟨*trs2mtrs sg* 159b⟩+≡

```
p_epic_spec  : Txt  -> Prog {external};
pp_epic_spec : Prog -> Txt  {external};
```

160c ⟨*trs2mtrs sg* 159b⟩+≡

```
ppp_trs2mtrs : Txt  -> Txt;
epic2mtrs    : Prog -> Prog;
trs2mtrs     : Mod -> Mod;
```

160d ⟨*trs2mtrs rules* 160a⟩+≡

```
ppp_trs2mtrs(Txt) = pp_epic_spec(epic2mtrs(p_epic_spec(Txt)));

epic2mtrs(Prog) =
  given e_at_m(e_subs_m(Prog)) as Mod  :
  given trs2mtrs(Mod)          as Mod' :
```

```
      mk_prog(Mod');

    trs2mtrs(Trs) =
      given complete(Trs)                                          as Trs'   :
      given t2m(e_subs_f(Trs'),e_subs_r(Trs'),e_subs_f(Trs'))      as Forms  :
      given cons(form_R(cat_id(mk_id(``),mk_id(`return`))),Forms) as Forms' :
        mk_mod(mk_cat(e_subs_f(Trs'),forms2t(Forms')),forms2e(Forms'));
```

160e ⟨*trs2mtrs.epg* 160e⟩≡

```
  module Trs2mtrs
  types
    ⟨yesno esg 128b⟩ ⟨id esg 130a⟩ ⟨char esg 128d⟩
    ⟨epic abstract syntax access 131b⟩ ⟨epic abstract syntax build 132c⟩
    ⟨mtrs forms esg 143d⟩ ⟨epic-utl esg 133a⟩
    complete     : Trs                  -> Trs {external};
    t2m          : Funs # Rules # Funs -> Forms {external};
    forms2t      : Forms                -> X {external};
    forms2e      : Forms                -> X {external};
    t2m_lhs      : Sq_r # Sq_f          -> List_Form {external};
    p_epic_spec  : Txt                  -> Prog {external};
    pp_epic_spec : Prog                 -> Txt  {external};
    ppp_trs2mtrs : Txt                  -> Txt;
    epic2mtrs    : Prog                 -> Prog;
    trs2mtrs     : Mod                  -> Mod;
  rules
  ⟨trs2mtrs rules 160a⟩
```

This code is written to file `trs2mtrs.epg`.

Appendix B

GEL binary encoding and C library

For a realistic implementation, a binary encoding of GEL was designed. In the binary encoding, numeric (binary) abbreviations are used instead of textual abbreviations, and a fixed format caters for fast reading.

For conversion to and from the textual format, there are tools `gel2a` (GEL to ascii) and `gel2b` (GEL to binary). Both tools have an option `-b` for an even older variant of the textual format that uses binary naturals in ascii (”0”, ”1”, ”10” etc.).

We will use the following notations in the definition of the binary format:

Notation	Meaning
`<n>`	A byte with value `n`.
`<'c'>`	A byte with the ASCII-value of the character `c`.
`xx<n>`	A byte starting (most significant bits) with `xx`, followed by a binary value `n` (remaining bits in the byte).
`#-k-`	A sequence of one or more bytes defining a positive number or index k, defined as follows: `0<n>` where $k = n$ and $(0 \leq k \leq 127)$ `10<n1> <n2>` where $k = 256 * n1 + n2$ and $(0 \leq k \leq 16383)$ `110<n1> <n2> <n3>` where $k = 256*(256*n1+n2)+n3$ and $(0 \leq k \leq 2M)$ `1110<n1> <n2> <n3> <n4>` where $k = 256*(256*(256*n1+n2)+n3)+n4$ and $(0 \leq k \leq 256M)$ etc. `11111110<n1> .. <n7>` for $(0 \leq k <= 2^{56} - 1)$
$$$	A byte-sequence `c1..ck`, encoded as: `#-k- <c1> <c2> ... <ck>` where `k` is the length of the sequence, followed by the sequence itself.

Given this notation, we shall now describe the binary format of GEL texts. A binary GEL text is an unseparated sequence of GEL items, where every item is either the introduction of an abbreviation, some stack operation, or a build command.

In the important case that the graphs are trees, the build command occurs most often, so it is important that the encoding of a build command is as short as possible. Build commands are encoded by a single number for the abbreviation (and an additional number for the arity in case of varyadic types), with the provision that the numbers 0... 10 are reserved as tags for the other, less frequently occurring commands.

B.0.5 Build commands

Encoding	Meaning
`#-k-`	$(10 < k < 2^{56} - 1)$, build node given the abbreviation no. `k` of a fixed arity type.
`#-k- #-l-`	$(10 < k < 2^{56} - 1)$, build node given the abbreviation no `k` of a varyadic arity type, with `l` children.

B.0.6 Abbreviations

Encoding	Meaning
`<0> #-s- #-k- $$$`	Introduction of abbreviation no. `s (s>10)` for fixed-arity type with `TYPE` $$$ and arity `k`
`<1> #-s- $$$`	Introduction of abbreviation no. `s (s>10)` with variadic arity and type $$$

B.0.7 Stack operations

Encoding	Meaning
`<2> #-k-`	duplicate stackitem at index `k` (top of stack is 0).
`<3> #-k-`	forward reference, index `k` (next node built is 0).
`<4> #-k-`	drop k items from stack.

B.0.8 Comments

Encoding	Meaning
`<5> $$$`	the comment $$$.

B.0.9 An example of a binary encoding

As an example, Figure B.1 on page 165 displays the binary encoding of the GEL text that was used as the running example in this document.

Binary Code	Description
`<0><11><0>`	Abbreviation 6 with sig 0
`<3><'O'><'N'><'E'>`	of type `ONE`.
`<1><12>`	Abbreviation 7 with sig *
`<4><'L'><'I'><'S'><'T'>`	of type `LIST`.
`<0><13><2>`	Abbreviation 8 with sig 2
`<4><'P'><'L'><'U'><'S'>`	of type `PLUS`.
`<11>`	build a node of type `ONE`
`<12><0>`	build a node of type `LIST` with 0 edges
`<13>`	build a node of type `PLUS`
`<2><0>`	reference top of stack
`<3><1>`	forward reference
`<13>`	build another `PLUS`
`<2><1>`	reference in stack
`<12><2>`	build a node of type `LIST` with 2 edges
`<5><1>`	drop one element from the stack

Figure B.1: A binary encoding of the running example

B.1 Using the C implementation of GEL: `<gel.h>`

We have implemented a C library of re-entrant functions for reading and writing of GEL representations; multiple graphs can be written concurrently. In most cases, this library will overcome the need to write an application specific reader or writer for GEL. The application specific part that needs to be written is typically a few pages of C.

To allow arbitrary internal representations of graphs for every application using the GEL library, the library does not maintain a complete representation of the graph, nor a complete representation of the abbreviation table. Instead, this is left to the application using the library.

For references to subgraphs of the graph maintained by the application, the GEL library uses a `C` type `gel_node`. During reading, when a new node must be built by the application, the edges are supplied by the library as an array of values of the `gel_node` type, and a reference to the new node is returned by the application as a new value of this type. Forward references are handled by supplying a new reference for a certain edge of an existing node. During writing, the GEL library traverses the application's graph by calling functions defined on values of the `gel_node` type.

For the implementation of the abbreviation table, the GEL library maintains a correspondence between `SHORT-TYPE`s and values of a C type `gel_type`, and the application maintains a correspondence between values of the type `gel_type` and corresponding `TYPE`s (sequences of bytes) and `SIG`s (integers, or varyadic). On reading an abbreviation, the GEL library asks the application to supply a `gel_type` for a combination of a `TYPE` and a

`SIG`. This `gel_type` is passed to the application when a new node must be built.

During writing, the GEL library may ask the application (several times) to supply the `TYPE` and the `SIG` belonging to a `gel_type`.

In Section B.1.1, we will describe the declarations pertinent to both reading and writing, in Section B.1.2 we will document the reading interface, and in Section B.1.3 we specify the writing interface.

B.1.1 General functionality

The file `<gel.h>` provides the following C types and macros:

```
typedef void *gel_node;
typedef long gel_type;  /* but not 0 */
#define VARYADIC -1
```

Usually, the tools will define private structures for nodes and type information, and use casts to convert to and from `gel_node` and `gel_type`. Arities are passed as integers, with `VARYADIC` denoting varyadic arities.

There is no predefined notion of characters in GEL. Because characters are used often in practice, the GEL library defines the following functions for the manipulation of 8-bit characters:

`char * char_name(int c)` Returns as a string a standard name for a character type ("'c", where c is the character). The names are statically allocated, so this function is cheaper than the obvious one-liner.

`bool is_char_name(char *s)` Checks if a string is a standard character name.

`int char_code(char *s)` Converts a character name into a character code.

Some tools might need a table to record the correspondence between `gel_type`s and type/signature combinations. If the types are C strings (ended by a NULL character), the following types and functions can be used for this purpose:

`typedef void *gel_table` Declare a variable of this type to hold the table, with initial value NULL. Allocation will be performed by the library.

`void gel_table_insert(gel_table table, char* type, gel_type gtype)` Insert into `table` the association between the full type-name `type` and gel-type `gel_type`. Note that `gel_type` should not be 0.

`gel_type gel_table_get(gel_table table, char *type)` Retrieves the `gel_type` associated with the full type-name `type`. Returns 0 if `type` is not present in the table.

`void gel_table_cleanup(gel_table table)` Reclaim space occupied by `table`.

Both `gel_table_insert` and `gel_table_get` use time linear in the length of the full typename.

Finally, for the implementation of a writer without using the library functions described in the next section, some basic functions for writing binary encodings of numbers and byte-sequences (see Appendix B) both to buffers and to files are supplied:

`void gel_bc_nat(long,FILE*)` writes a number to file.

`char *gel_bc_snat(long,char*)` writes a number to a buffer, and returns a pointer into the buffer, just after the last byte written.

`void gel_bc_str(int sz, char *bytes, FILE *f)` writes the byte-sequence with length code, length `sz` bytes, to file.

`char *gel_bc_sstr(int sz, char *bytes, char *buf)` writes the length-encoded byte-sequence, length `sz` bytes, to a buffer. Returns a pointer into the buffer, just after the last byte written.

B.1.2 Using the GEL reader

There are two functions for reading a GEL text, one for reading from file, the other for reading from a buffer:

```
gel_node gel_read(FILE*,gel_rcfg)
gel_node gel_sread(char*,int,gel_rcfg)
```

The first argument of `gel_sread` is the buffer, and the second argument is the number of characters in the buffer. The last argument of both functions is a pointer to a structure containing pointers to some application-specific functions (a GEL Read ConFiGuration, hence the name of the `C` type),

```
   typedef struct _gel_rcfg {
           gel_type (*define)    (int,int,char*);
           gel_node (*build)     (gel_type,int,gel_node*);
/* OPT */  void     (*set)       (gel_type,gel_node,int,gel_node);
/* OPT */  void     (*error)     (char *msg);
   } *gel_rcfg;
```

where `/* OPT */` indicates that the `set` function is optional. Undefined function pointers should be set to `NULL`.

The application-specific callback functions should do the following:

`gel_type define(int ar, int tsize, char* type);` Returns the `gel_type` with external type representation `type` of size `tsize` and arity `ar`.

`gel_node build(gel_type i, int n, gel_node child[]);` Returns a node with arity `n` and children `child[0]... child[n-1]`. `i` tells what type of node should be built. If some child `x` is a forward reference (eg. cycle), `child[x]` is `NULL`, and later the call `set(i,t,x,s)` will be done, where `t` is the node returned by this instance of `build` and `s` is the real son.

`void set(gel_type i, gel_node t, int n, gel_node s); /* OPTIONAL */` Set child `#n` of term `t` to the term `s`.

`void error(char *msg); /* OPTIONAL */` Print the error message, and clean up.

B.1.3 Using the GEL writing algorithm

There are two functions for writing a GEL text, one for writing to file, the other for writing to a buffer (which returns the number of characters written).

```
void gel_write(FILE*,gel_node,gel_wcfg);
int  gel_swrite(char*,gel_node,gel_wcfg);
```

The first argument is either the file or the buffer to write to, the second argument is the root node of the graph, and the third argument is a pointer to a structure containing pointers to application-specific functions (a GEL Write ConFiGuration, hence the name of the `C` type),

```
        typedef struct _gel_wcfg {
                gel_type (*type)      (gel_node);
                int      (*arity)     (gel_type);
                int      (*tsize)     (gel_type);
                char    *(*trepr)     (gel_type);
                int      (*size)      (gel_node);
                gel_node (*child)     (gel_node,int);
     /* OPT */  void     (*error)     (char*);
     /* OPT */  int      (*unshare)   (gel_node);
     /* OPT */  int      (*compress)  (gel_node);
        } *gel_wcfg;
```

where `/* OPT */` indicates that a function is optional. The callback functions in this struct should be implemented as:

`gel_type type(gel_node t);` return the `gel_type` of a node.

`int arty(gel_type t);` return the arity of a `gel_type`, where the macro `VARYADIC` is defined -1 to mean varyadic arities.

`int tsize(gel_type t);` return the size of the byte representation of the type. If the representation of a type is a string (not containing `NULL` characters) this can be implemented as `strlen(trepr(t))`.

`char *trepr(gel_type t);` return the address of a buffer containing the byte representation of a type. In many cases, this will simply be an ASCII string.

`int size(gel_node t);` return the actual number of children of `t`.

`gel_node child(gel_node t,int i);` return the `i`th child of t.

`void error(char* msg); /* OPTIONAL */` Print the message `msg` and clean up.

`int unshare(gel_node n); /* OPTIONAL */` if defined, should return non-zero value if it is allowed to unshare the node `n`.

`int compress(gel_node n); /* OPTIONAL */` if defined should return non-zero value if it is allowed to compress the node `n`.

B.1.4 A complete example of a tool

A simple example of the use of our library is given below. This example is able to read and write binary trees with internal nodes typed `AND`, and leaves typed `TRUE` or `FALSE`.

```
#include "gel.h"

/* codes for the node types */
#define TRUE  1
#define FALSE 2
#define AND   3

/* table of types and names */
gel_table bool_types;

/* initialize table */
void init_table()
{ gel_table_insert(&bool_types,"TRUE",TRUE);
  gel_table_insert(&bool_types,"FALSE",FALSE);
  gel_table_insert(&bool_types,"AND",AND);
}

/* define a structure for nodes */
typedef struct _node {
        gel_type code;
        gel_node *children;
} *node;

/* functions for reading */
static void error(char *msg)
{ fprintf(stderr,"%s\n",msg);
  exit(1);
}
```

```
gel_type define(int ar, int tsize, char* type)
{  gel_type t;
   if (t= gel_table_get(&bool_types,type))
      return t;
   else error("unknown type");
}

gel_node build(gel_type t, int ar,
               gel_node child[])
{  node n = (node)
     calloc(1,sizeof(struct _node));
   switch(t) {
     case TRUE: n->code = TRUE;
                return (gel_node)n;
     case FALSE: n->code = FALSE;
                 return (gel_node)n;
     case AND: n->code = AND;
               n->children = (gel_node *)
                  malloc(2*sizeof(gel_node));
               n->children[0] = child[0];
               n->children[1] = child[1];
               return (gel_node)n;
     default: error("unknown type");
   }
}
```

```
static struct _gel_rcfg RC = {
  define, build, NULL, error
};

/* functions for writing */
static gel_type type(gel_node n)
{return ((node)n)->code;}

static int arity(gel_type t)
{ switch(t)
  { case TRUE:  return 0;
    case FALSE: return 0;
    case AND:   return 2;
    default: error("internal error");
  }
}

static int tsize(gel_type t)
{ switch(t)
  { case TRUE:  return 4;
    case FALSE: return 5;
    case AND:   return 3;
    default: error("internal error");
  }
}

static char *trepr(gel_type t)
{ switch(t)
  { case TRUE:  return "TRUE";
    case FALSE: return "FALSE";
    case AND:   return "AND";
    default: error("internal error");
  }
}
```

```
static int size(gel_node n)
{ return arity(((node)n)->code); }

static gel_node child(gel_node n, int c) {
  switch(((node)n)->code) {
    case TRUE: error("no children");
    case FALSE: error("no children");
    case AND: if (c<2)
                 return((node)n)->children[c];
              else error("wrong index");
    default: error("internal error");
  }
}

static struct _gel_wcfg WC = {
  type, arity, tsize, trepr, size,
  child, error, NULL, NULL
};

main()
{ gel_node n;

  /* initialize */
  init_table();

  /* first read a graph, then write it */
  n= gel_read(stdin,&RC);
  gel_write(stdout,n,&WC);

  exit(0);
}
```

B.2 Full-Gel-syntax

Module Full-Gel-syntax
imports Gel$^{(7.7.1)}$
exports
  lexical syntax
    ~[\n]+ → TYPE
    "\␣" → SPACE
    ['*a-zA-Z0-9*] → SHORT-TYPE
    "\n" → RET

Scanners generated from SDF definitions always prefer non-Layout. Effectively, the very liberal lexical sorts defined in this module would prevent the recognition of Layout between the equations of other modules. Therefore, it is not possible to define these lexical sorts in the module Gel-types.

B.3 Layout

```
Module Layout
exports
  lexical syntax
    [␣\t\n]             → LAYOUT
    “//”~[\n]*[\n]      → LAYOUT
    “%%”~[\n]*[\n]      → LAYOUT
```

Bibliography

[Amt93] Torben Amtoft. Minimal thunkification. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1993.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BBKW89] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(1):283–301, 1989.

[BC93] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[BDHF96] J.A. Bergstra, T.B. Dinesh, J. Heering, and J. Field. A complete transformational toolkit for compilers. In Hanne Riis Nielson, editor, *Programming Languages and Systems – ESOP'96*, number 1058 in Lecture Notes in Computer Science, pages 92–107. Springer-Verlag, 1996.

[Ben93] P.N. Benton. Strictness properties of lazy algebraic datatypes. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 206–217. Springer-Verlag, 1993.

[BEØ93] Didier Bert, Rachid Echahed, and Bjarte M. Østvold. Abstract rewriting. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, 1993.

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BM92] Geoffrey Burn and Daniel Le Métayer. CPS-translation and the correctness of optimising compilers. Technical Report DoC92/20, Imperial College, Department of Computing, 1992.

[Bou94] Adel Bouhoula. Sufficient completeness and parameterized proofs by induction. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '94*, 1994.

[BS96] J.A. Bergstra and M.P.A. Sellink. Sequential data algebra primitives. Technical Report P9602, University of Amsterdam, Programming Research Group, march 1996.

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, 1991.

[BvEJ$^+$87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W.Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In J.W. de Bakker, A.J. Nijman, and vol. II P.C. Treleaven, editors, *Proceedings PARLE'87 Conference*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Verlag, 1987.

[CCI] CCITT. Specification of basic encoding rules for abstract syntax notation one (asn.1). Recommendation X.209, technically aligned with ISO 8825.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.

[CH90] Guy Cousineau and Gérard Huet. The CAML primer. Technical report, Inria, 1990. Version 2.6.1, available by ftp from ftp.inria.fr.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT press, Cambridge Massachussetts, 1989.

[DF95] Rémi Douence and Pascal Fradet. A taxonomy of functional language implementations. Technical Report 972, IRISA, december 1995. Part I: Call-by-Value.

[DJ90] N. Dershowitz and J.-P Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol B.*, pages 243–320. Elsevier Science Publishers, 1990.

[DR94] A.M. Dery and L. Rideau. A message protocol for distributed programming environments. Technical report, GIPE II, Generation of Interactive Programming Environments, phase 2, sixth review report, january 1994. ESPRIT project 2177.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, Vol. I, Equations and Initial Semantics.* Springer-Verlag, 1985.

[ESL89] H. Emmelmann, F-W. Schröer, and R. Landwehr. BEG - a Generator for Efficient Back Ends. In *Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation.* ACM Press, July 1989. Appears as SIGPLAN Notices, Vol. 24, Number 7.

[Fie90] J. Field. On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis. In *Proc. ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[FM91] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, january 1991.

[Fra94] Ulrich Fraus. Inductive theorem proving for algebraic specifications - TIP system user's manual -. Technical report, Passau, 1994.

[FvdP96] Wan Fokkink and Jaco van de Pol. Correct transformation of rewrite systems for implementation purposes. Technical Report 164, Utrecht University Logic Group Preprint Series, may 1996.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer-Verlag, 1987.

[GG91] S.J. Garland and J.V. Guttag. *A Guide to LP, The Larch Prover.* MIT, November 1991.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GH94] Robert Giegerich and John Hughes. Functional programming in the real world. Technical Report 89, Schloss Dagstuhl, Germany, 1994.

[GL91] Bernhard Gramlich and Wolfgang Lindner. A guide to unicom, an inductive theorem prover based on rewriting and completion techniques. Technical Report SR-91-17, Fachbereich Informatik, Universität Kaiserslautern, 1991. SEKI Report.

[GWM$^+$92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In J.A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ.* Cambridge University Press, 1992. To Appear.

[Hen89] P.R.H. Hendriks. Lists and associative functions in algebraic specifications - semantics and implementation. Report CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.

[HF$^+$96] Pieter H. Hartel, Marc Feeley, et al. . Benchmarking implementations of functional languages with “pseudoknot”, a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[HG94] Lutz H. Hamel and Joseph A. Goguen. Towards a provably correct compiler for OBJ3. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '94*, 1994.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HJW$^+$92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, May 1992.

[HL91a] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems part I and II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic; essays in honour of Alan Robinson*, pages 395–443. MIT Press, 1991.

[HL91b] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, I. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, chapter 11. MIT Press, Cambridge, Massachusetts, 1991.

[HM93] B.M. Hearn and K. Meinke. ATLAS: A type language for algebraic specification. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, number 816 in Lecture Notes in Computer Science, pages 146–168. Springer-Verlag, 1993.

[HO82] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[Ing61] P.Z. Ingermann. Thunks – a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[JL92] Simon L Peyton Jones and David Lester. *Implementing Functional Languages – A Tutorial.* Prentice Hall, 1992.

[Jr.85] R.H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM transactions on programming languages and systems*, 7(4):501–538, october 1985.

[JS89] Simon L Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Functional Programming and Computer Architecture*, pages 184–201. ACM, 1989.

[Kam94a] J.F.Th. Kamperman. GEL, a graph exchange language. Report CS-R9440, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994. Available by ftp from ftp.cwi.nl:/pub/gipe as Kam94.ps.Z.

[Kam94b] J.F.Th. Kamperman. Gel, a graph exchange language. In R.Giegerich and J.H.Hughes, editors, *Functional programming in the Real World*, volume 89 of *Dagstuhl Seminar Report.* Schloss Dagstuhl, 1994. Abstract of lecture.

[KDW94] J.F.Th. Kamperman, T.B. Dinesh, and H.R. Walters. An extensible language for the generation of parallel data manipulation and control packages. In Peter A. Fritzson, editor, *Proceedings of the Poster Session of Compiler Construction '94*, 1994. Appeared as technical report LiTH-IDA-R-94-11, university of Linköping.

[KKSdV93] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice.* John Wiley & Sons Ltd, 1993.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

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

[KW93a] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by ftp from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[KW93b] J.F.Th. Kamperman and H.R. Walters. ARM – Abstract Rewriting Machine. In H.A. Wijshoff, editor, *Computing Science in the Netherlands*, pages 193–204, 1993.

[KW95a] J.F.Th. Kamperman and H.R. Walters. Lazy rewriting and eager machinery. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, number 914 in Lecture Notes in Computer Science, pages 147–162. Springer-Verlag, 1995.

[KW95b] J.F.Th. Kamperman and H.R. Walters. Minimal term rewriting systems. Technical Report CS-R9573, CWI, december 1995. Available as http://www.cwi.nl/epic/articles/CS-R9573.ps.Z. To appear in the proceedings of the 11th Workshop on Abstract Data Types, published by Springer-Verlag.

[KW96] J.F.Th. Kamperman and H.R. Walters. Simulating TRSs by Minimal TRSs: a simple, efficient, and correct compilation technique. Technical Report CS-R9605, CWI, january 1996. Available as http://www.cwi.nl/epic/articles/CS-R9605.ps.Z.

[KZ89] Deepak Kapur and Hantao Zhang. RRL: Rewrite rule laboratory user's manual. Technical Report 89-03, The University of Iowa, 1989.

[Lam83] D.A. Lamb. *Sharing Intermediate Representations: The Interface Description Language.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1983.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 121(2):214–233, september 1995.

[Mar92] Luc Maranget. *La stratégie paresseuse.* PhD thesis, Université Paris VII, July 1992.

[Mid90] A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[Mil84] R. Milner. A proposal for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197. ACM, 1984.

[MMG94] M.Feeley, M.Turcotte, and G.Lapalme. Using multilisp for solving constraint satisfaction problems: an application to nucleic acid 3d structure determination. *Lisp and symbolic computation*, 7(2/3):231–246, 1994.

[MMR86] A. Martelli, C. Moiso, and C.F. Rossi. An algorithm for unification in equational theories. In *Proceedings of the Symposium on Logic Programming*, pages 180–186. IEEE Computer Society, 1986.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[N+78] J.M. Newcomer et al. PQCC implementor's handbook. Technical report, CMU, 1978. Internal Report.

[New87] Joseph M. Newcomer. Efficient binary i/o of IDL objects. *SIGPLAN Notices*, 22(11):35–43, nov 1987.

[O'D77] M.J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7:57–82, 1994.

[Pet92] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, editors, *Proceedings of the Fourth International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 258–270. Springer-Verlag, 1992.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[PvE93] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[Ram92] Norman Ramsey. Literate programming tools need not be complex. Technical Report TR-351-91, Department of Computer Science, Princeton University, October 1991, revised September 1992.

[RC91] J.A. Rees and W. Clinger. Revised report on the algorithmic language scheme. Technical report, MIT, november 1991.

[Sno89] Richard Snodgrass. *The Interface Description Language, Definition and Use.* Principles of Computer Science Series. Computer Science Press, Rockville, MD 20850, 1989.

[SW94] Paul Steckler and Mitchell Wand. Selective thunkification. In *First International Static Analysis Symposium*, Namur, Belgium, 28-30 September 1994. also available by ftp as sas94.ps.Z from ftp.ccs.neu.edu:/pub/people/steck.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[vdBV96] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1), 1996.

[vDHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Inc., april 1996.

[vdP96] Jaco van de Pol. Operational semantics of term rewriting with priorities. Technical Report 162, Utrecht Research Institute for Philosophy, april 1996.

[Ver95] Rakesh M. Verma. A theory of using history for equational systems with applications. *Journal of the ACM*, 42(5):984–1020, september 1995.

[Vis96] Eelco Visser. Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Inc., april 1996.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the fourteenth ACM conference on Principles Of Programming Languages*, pages 307–313, 1987.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. Available by ftp from ftp.cwi.nl:/pub/gipe/reports as Wal91.ps.Z.

[Wal94] H.R. Walters. A complete term rewriting system for decimal integer arithmetic. Technical Report CS-9435, Centrum voor Wiskunde en Informatica, 1994. Available by ftp from ftp.cwi.nl:/pub/gipe as Wal94.ps.Z.

[WK94] H.R. Walters and J.F.Th. Kamperman. A hybrid interpreter for arm term rewriting. Compare deliverable CWI-0002-arm, CWI, 1994.

[WK96a] H.R. Walters and J.F.Th. Kamperman. Epic 1.0 (unconditional), an equational programming language. Technical Report CS-R9604, CWI, january 1996. Available as http://www.cwi.nl/epic/articles/epic10.ps.

[WK96b] H.R. Walters and J.F.Th. Kamperman. Epic: An equational language – abstract machine and supporting tools. In Harald Ganzinger, editor, *Rewriting Techniques and Applications 1996*, Lecture Notes in Computer Science. Springer-Verlag, 1996. to appear.

[WZ95] H.R. Walters and H. Zantema. Rewrite systems for integer arithmetic. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, number 914 in Lecture Notes in Computer Science, pages 324—338. Springer-Verlag, 1995.

Samenvatting

Het manipuleren van formules is een cruciale ingrediënt van zowel wiskunde als informatica. Een voorbeeld hiervan is het gebruik van de *vergelijking* voor het zogenaamde merkwaardig product

$$(a+b)\cdot(a-b)=a^2-b^2, \tag{B.1}$$

om uit het hoofd uit te rekenen dat:

$$21\cdot 19=(20+1)\cdot(20-1)=20^2-1^2=400-1=399.$$

In vergelijking (B.1) worden de symbolen $+$, $\cdot$, $-$ en ^{2} gebruikt om *operaties* aan te duiden, en de symbolen a en b als *variabelen* waarvoor willekeurige getallen kunnen worden ingevuld. Ook de *constanten* zoals 1 en 20 die in de berekening worden gebruikt zijn feitelijk symbolen. De uit symbolen opgebouwde expressies zoals 1, $a+b$ en $(a-b)$ worden *termen* genoemd.

Toepassingen van symbolische manipulatie beperken zich niet tot rekenen alleen. In de taalkunde kan het feit dat de zin "Krajicek wint de wedstrijd" hetzelfde betekent als de zin "De wedstrijd wordt door Krajicek gewonnen" worden uitgedrukt met de regel

$$x \textbf{ wint } y = y \textbf{ wordt door } x \textbf{ gewonnen} \tag{B.2}$$

Typerend voor vergelijkingen zoals (B.1) en (B.2) is dat ze, afhankelijk van de situatie, zowel van links naar rechts als van rechts naar links kunnen worden toegepast. Welke richting gekozen moet worden is echter niet altijd duidelijk. Soms is het nodig een relatief eenvoudige term ingewikkelder te maken voordat een beslissende vereenvoudiging kan worden bereikt. Kortom, voor de efficiënte toepassing van vergelijkingen is intelligentie nodig. Dit maakt vergelijkingen ongeschikt voor directe verwerking door computers.

Herschrijfregels zijn vergelijkingen die alleen van links naar rechts gebruikt mogen worden. Omdat de richting vastligt zijn herschrijfregels veel eenvoudiger te verwerken dan vergelijkingen. Een verzameling van herschrijfregels wordt een *termherschrijfsysteem* genoemd. In een termherschrijfsysteem van twee regels kan bijvoorbeeld optelling van natuurlijke getallen gedefinieerd worden:

$$\begin{aligned} 0+x &\rightarrow x && \text{(B.3)}\\ \mathrm{s}(x)+y &\rightarrow \mathrm{s}(x+y). && \text{(B.4)} \end{aligned}$$

In dit voorbeeld staat het symbool s voor 'successor' ofwel 'opvolger', zodat de natuurlijke getallen $0,1,2,3\ldots$ worden voorgesteld door de termen $0,\mathrm{s}(0),\mathrm{s}(\mathrm{s}(0)),\mathrm{s}(\mathrm{s}(\mathrm{s}(0))),\ldots$. Met het bovenstaande termherschrijfsysteem is eenvoudig te berekenen dat $1+1=2$:

$$\mathrm{s}(0)+\mathrm{s}(0)\rightarrow\left\{\begin{array}{c}\text{gebruik (B.4) met}\\ x=0 \text{ en } y=\mathrm{s}(0)\end{array}\right\}\rightarrow \mathrm{s}(0+\mathrm{s}(0))\rightarrow\left\{\begin{array}{c}\text{gebruik (B.3) met}\\ x=\mathrm{s}(0)\end{array}\right\}\rightarrow \mathrm{s}(\mathrm{s}(0))$$

Het toepassen van een regel heet een *herschrijfstap*. In dit geval is er bij iedere stap steeds maar één regel die kan worden toegepast, maar in het algemeen hoeft dit niet het geval te zijn. Een *strategie* geeft aan welke regel gekozen moet worden als er meerdere mogelijkheden zijn. In combinatie met een strategie kan een herschrijfsysteem gezien worden als een rekenrecept voor een klasse van berekeningen. In de informatica wordt een dergelijk rekenrecept een *programma* genoemd.

Hoewel de naam 'computer' (rekenaar) voor een apparaat al aangeeft dat het gemaakt is om berekeningen uit te voeren, is de afstand tussen termherschrijfsystemen en programma's die rechtstreeks door een computer kunnen worden uitgevoerd nogal groot. Het is zeer bewerkelijk om met de hand een door een computer uitvoerbaar programma te construeren voor een bepaalde berekening. Daarom gebruikt men al sinds de jaren vijftig vertaalprogramma's die een voor mensen begrijpelijke versie (in ons geval het termherschrijfsysteem) vertalen naar een voor computers verwerkbare versie. Een direct verwerkbare versie van een programma wordt een *executeerbaar* programma genoemd.

Net zoals er voor de vertaling van *natuurlijke taal* zowel (simultaan)tolken als vertalers worden ingezet, zijn ook computervertaalprogramma's te onderscheiden in *interpreters* (tolken) en *compilers* (vertalers). Een interpreter vertaalt, afhankelijk van de actuele situatie, steeds het benodigde stukje van het programma, waarna het vertaalde stukje door de computer wordt uitgevoerd. Een compiler vertaalt daarentegen in één keer het gehele programma, waarna het talloze malen kan worden uitgevoerd.

Iedere vertaling kost tijd. Voor interpreters levert dit een efficiëntie-nadeel op omdat sommige stukken vele malen worden uitgevoerd en daarom evenzovele malen moeten worden vertaald. Een compiler heeft het relatieve nadeel dat men eerst moet wachten tot het hele programma is vertaald. Dit nadeel laat zich vooral voelen tijdens het ontwikkelen van programma's.

Het in dit proefschrift beschreven onderzoek is verricht in een groep die werkt aan een systeem voor het ontwerpen en prototyperen van software voor symbolische manipulatie. Het zal de lezer niet verbazen dat dit systeem gebaseerd is op termherschrijven. Het systeem bevat dan ook een snelle interpreter voor termherschrijfsystemen. De kwaliteit van dit systeem is inmiddels dusdanig, dat er bij de academische en industriële gebruikers een behoefte is ontstaan om de ontworpen en geprototypeerde programma's sneller, en liefst onafhankelijk van het systeem, te kunnen uitvoeren. Dit vormt de belangrijkste motivatie voor het onderzoek in dit proefschrift, dat de *compilatie van termherschrijfsystemen* behandelt.

De belangrijkste eis die aan compilatie wordt gesteld is *correctheid*. Correctheid wil in dit geval zeggen dat de vertaling die door de computer wordt uitgevoerd hetzelfde resultaat oplevert als het oorspronkelijke termherschrijfsysteem.

Omdat de afstand tussen termherschrijfsystemen en executeerbare programma's zo groot is, is in dit proefschrift de vertaling in twee stappen opgesplitst. In de eerste stap worden termherschrijfsystemen vereenvoudigd naar *minimale* termherschrijfsystemen. In de tweede stap worden minimale termherschrijfsystemen omgezet in executeerbare programma's.

De meeste energie is gestoken in het ontwerp van minimale termherschrijfsystemen. Enerzijds moet deze deelklasse van termherschrijfsystemen zo ruim zijn dat elke berekening erin uitgedrukt kan worden, anderzijds moeten minimale termherschrijfsystemen een eenvoudige vertaling naar een executeerbare versie toelaten. Dit laatste is zo goed gelukt, dat de tweede stap, en daarmee de correctheid ervan, zeer eenvoudig is.

De eerste stap is veel gecompliceerder, maar is uit te voeren omdat hij zich geheel in de wereld van termherschrijfsystemen afspeelt. Om het correctheidsbewijs ervan te kunnen leveren, is eerst theorie ontwikkeld over *simulatie* van het ene termherschrijfsysteem door het andere. Met behulp van deze theorie is de correctheid van de eerste stap bewezen.

Het zal de lezer wellicht niet verbazen dat de vertaling van algemene termherschrijfsystemen naar minimale termherschrijfsystemen zelf *ook* opgevat kan worden als een termherschrijfsysteem. In hoofdstuk 5 wordt er volgens dit inzicht een termherschrijfsysteem gepresenteerd dat termherschrijfsystemen (ook zichzelf !) naar minimale termherschrijfsystemen kan vertalen.

In de hierboven genoemde hoofdstukken is steeds uitgegaan van een vaste strategie, namelijk het herschrijven van de binnenste, meest rechtse herschrijfbare deelterm. Hoewel het herschrijven van binnenste deeltermen het meest efficiënt is per herschrijfstap, zijn er herschrijfsystemen waarbij deze strategie leidt tot onnodig veel stappen. Een gedeeltelijke oplossing van dit probleem is *luie evaluatie*, een strategie waarbij stappen die potentiëel niet nodig zijn zo veel mogelijk worden uitgesteld.

Hoofdstuk 6 laat zien dat luie evaluatie grotendeels kan worden bereikt door een vertaling op het niveau van termherschrijfsystemen. Er blijken nauwelijks aanpassingen nodig te zijn aan de machinerie die ontwikkeld is voor het herschrijven van binnenste deeltermen in minimale termherschrijfsystemen.

Omdat programma's vaak uit meerdere delen zijn samengesteld, is het van belang dat er tussen deze delen efficiënte informatie-uitwisseling mogelijk is. In Hoofdstuk 7 wordt een taal gedefiniëerd voor het gecomprimeerd uitwisselen van termen. In deze taal is het mogelijk lange symbolen te vervangen door afkortingen. Bovendien hoeven deeltermen die vaker voorkomen slechts één keer te worden beschreven. Deze taal wordt onder andere gebruikt om termherschrijfsystemen aan de in dit proefschrift beschreven compiler aan te bieden.

Tenslotte wordt in Hoofdstuk 8 verslag gedaan van enkele metingen die aantonen dat de in dit proefschrift beschreven technieken efficiënte programma's opleveren.