Compiling a Functional Language to Efficient SACLIB C[*]

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
Email: `Wolfgang.Schreiner@risc.uni-linz.ac.at`

September 20, 1993

Abstract

In this paper, we describe the design and implementation of D, a small functional language with strict execution semantics. We define the transformation and compilation techniques that we apply to translate a D program into portable C code that corresponds (with some restrictions) in structure and quality to a program written by an experienced human programmer. The target code may use the runtime system of the computer algebra library SACLIB. *Thus a D program can call all functions of this library and can be itself easily embedded into* SACLIB *applications. D is going to serve as the basis of a parallel functional programming system for computer algebra.*

[*]Supported by the FWF grant S5302-PHY "Parallel Symbolic Computation".

Contents

1 Introduction

In this paper, we describe the design and implementation of D, a small functional language with strict execution semantics. In particular, we define the transformation and compilation techniques that we apply to translate a D program into portable C code that corresponds (with some restrictions) in structure and quality to a program written by an experienced human programmer. The generated code can be easily embedded into other C programs.

The background of our work is as follows. For two years, the parallel computation group at RISC-Linz has pursued the design and implementation of a library of parallel computer algebra algorithms for shared memory multi-processors. Based on SACLIB, a library of C functions for computer algebra [BCE+92], we have developed the PACLIB kernel, a parallel variant of the SACLIB runtime system (supporting memory management and garbage collection) with extensions for parallel C programming [HSN+92].

Using the existing SACLIB library and the new PACLIB kernel, we have implemented a variety of parallel computer algebra algorithms (linear equation solving, Gröbner bases computation, polynomial resultant computation, p-adic arithmetic, ...). Unfortunately it has turned out that the mental and technical overhead for parallel programming in PACLIB is rather large, which considerably limits productivity.

The development of a new PACLIB algorithm consists in the study of the mathematical knowledge, its inherent algorithmic ideas and their implementation in a SACLIB program. With this background a parallel algorithm is developed and implemented in a PACLIB program. After testing and debugging the parallel program, its performance is evaluated and often turns out to be insufficient.

In this case, an iterative process begins that consists of the analysis of the program for parallelism bottlenecks, the restructuring of the program, the debugging of the new program, and its evaluation. Already small changes in the computational structure of the basic algorithm may have significant impacts on the performance of the program. However, even slight algorithmic modifications usually result in a large programming overhead.

The main reason for this is that the level of abstraction in an imperative program, in particular in a parallel one, is rather low. The actual computational pattern of the algorithm, i.e. the interaction between program parts and thus the communication and synchronization between tasks, is overlaid by much technical overhead (order of statements, introduction of auxiliary variables, ...). This makes experiments with different parallel variants of a program very tedious and may result in suboptimal solutions to be accepted.

Consequently we want to increase the level of parallel programming abstraction. We pursue an approach that is based on a functional notation where the programmer can specify a parallel algorithm in various levels of abstractions (from explicit parallelization annotations to algebraic properties that the compiler exploits for restructuring). We expect that the compact notation and the simple transformation rules of functional languages will more clearly expose the computational structure of a program, better support the development of parallelization ideas, and allow program manipulations with less mental overhead.

However, there are some side conditions for this approach. First we are not willing to significantly sacrifice performance for simplicity. Since we put much effort in improving a speedup of say 12 to a speedup of 15 on a 20 processor machine, it is not acceptable if the basic overhead for a functional program lets it run two times slower than a corresponding imperative one. Moreover most SACLIB algorithms do not destructively update objects but compose them in a constructive manner. *Hence there is no reason why a corresponding implementation in a functional language should be less efficient.*

Second, we would like to freely mix library modules implemented in the functional language with library modules directly implemented in C. We still want to have the possibility to write time-critical code in C, and, more important, the generated code shall be called from any C program. Remember we want to develop a *library* to be used by application programs, not a stand-alone system. *The functional layer shall be transparent to the imperative layer.*

Currently, no existing implementation of a functional language fulfills both demands (actually, we are not aware of any implementation that totally fulfills the second demand). Consequently, we have designed our own functional language and written a compiler for it. This paper describes this language and its implementation.

From the presentation above it should be clear that our motivation for thia work was not to have another functional language with many new concepts. In fact, D contains only those features that we considered essential for our applications and that we could compile into efficient C code. Other features (general higher order functions, algebraic data types, overloading, modules, input/output) were deliberately omitted. However, the concept of *iterated compositions* was introduced to express typical program patterns more concisely.

The *original* parts of this paper are

1. *Computational Patterns:* The treatment of higher-order functions (Subsections 4.1 and 4.2) and the introduction of iterated compositions (Section 3) as compact abbreviations for first-order computational patterns.
2. *High-Level Imperative Target:* The generation of high-level imperative code (Section 5) from a functional program that can be called from external C programs and that may itself contain calls of external C functions (Section 6).
3. *High Code Quality:* Several optimizations that transform typical recursive function patterns into a tail-recursive form that is further transformed into efficient loop iteration (Subsections 5.3 and 5.4).

We believe that this work shows how many features of existing functional languages can be viewed as abbreviations for corresponding structures in imperative languages and can thus be efficiently compiled into imperative form.

Our *actual goal* is to have a platform for our ongoing work on a parallel functional programming system on top of PACLIB. Consequently, the reference implementation of the runtime system for the D language is based on the SACLIB kernel. However, the generated code is independent of the underlying runtime system and can be adapted to other systems as well.

2 The Core Language

The D language and its implementation provide the following features:

1. *Referential Transparency:* A D program is an expression whose evaluation does not cause any side effects. Consequently, there exist simple rules for reasoning and manipulation which are exploited by our compiler.
2. *Strict Semantics:* D has a strict evaluation semantics i.e. the arguments of a function can be evaluated before the function is called. This allows an efficient implementation of the language and a simple reasoning about its resource requirements.
3. *Polymorphic Type System:* The language allows to define functions that operate on objects of various types of a given pattern. The compiler translates a polymorphic function into a set of monomorphic ones.
4. *Function Templates:* D allows a limited class of higher-order functions, namely functions that are parameterized by other functions. Such functions are instantiated at compile-time yielding various first-order functions.
5. *Iterated Compositions:* D provides a flexible iteration construct that allows to express in a concise way (without applying recursive functions) frequently occurring computational patterns.

6. *C Target Code:* The compiler generates standard C code from a D program. With some restrictions, this C code corresponds in structure and quality to the C code written by an experienced human programmer.

7. *External Interface:* The generated code can be called from other C programs. Moreover, a D program can call external C functions as well.

In this section we formally define the semantics of a core of this language. Sections 3, 4, and 6 describe how the remaining features of D are translated into this core language.

For the formal part, we essentially apply the denotational framework described in [Sch86].

2.1 Abstract Syntax

Figure 1 gives the abstract syntax of the core language. The abstract syntax does *not* necessarily correspond to the concrete textual syntax of the full language (which loosely reminds of HASKELL and is given in Appendix A) but rather reflects the structure of the parse tree in the compiler. Keywords are only used as syntactic sugar to improve legibility of the definition.

This abstract syntax reflects the structure of the program after several phases of program transformations:

1. *Uncurrying:* Each partial application of a function (with less actual arguments than formal parameters) has been replaced by the identifier of a new function whose arity is the number of missing arguments.

2. *Specialization:* Each application of a higher-order function has been transformed into the application of an instance of this function by updating the formal parameter function with the actual argument function.

3. *Monomorphization:* Each application of a polymorphic function has been transformed into the application of a monomorphic instance of this function.

4. *Lambda Lifting:* All function definitions have been transformed into combinators (i.e. functions that do not reference the global environment) and lifted to the top-level.

5. *Type Annotations:* Type definitions and signatures have been removed and each function definition has been annotated by its type. We also assume all constant definitions to be annotated (just for uniformity, see Subsection 2.7).

The core therefore represents the first-order monomorphic subset of the language where functions only interact by their arguments. In Section 4, we will describe the translation of the full language into this form.

This section does not deal with the semantics of composition expressions (treated in Section 3) and of the external program interface (described in Section 6).

2.2 Notation

In the forthcoming denotational definitions, we will use two special constants:

$$\begin{aligned} \perp &= \text{“non-termination”} \\ \underline{\perp} &= \text{“error”} \end{aligned}$$

$\perp$ denotes the non-termination of a program, while $\underline{\perp}$ denotes some program error. It is common practice to use $\perp$ for both purposes as well, but we feel that the significant operational difference between a program that does not terminate and a program that triggers an error also deserves a denotational distinction. However we do not distinguish

program	$\rightarrow$	`let` *fdefs* `in` *exp*	Function Definitions
exp	$\rightarrow$	`let` *cdefs* `in` *exp*	Constant Definitions
	$\mid$	`if` exp_c `then` exp_t `else` exp_f	Conditional
	$\mid$	`case` exp_s `of` *patterns*	List Selection
	$\mid$	exp_h `:` exp_t	List Construction
	$\mid$	*exp* `.` *integer*	Tuple Selection
	$\mid$	`(` exp_1 `,` ... `,` exp_n `)`	Tuple Construction
	$\mid$	*id* `(` exp_1 `,` ... `,` exp_n `)`	Function Application
	$\mid$	exp_1 *binop* exp_2	Binary Expression
	$\mid$	*unop exp*	Unary Expression
	$\mid$	*const*	Constant
	$\mid$	*id*	Variable
patterns	$\rightarrow$	`[ ] =>` exp_n `;` id_h `:` id_t `=>` exp_c	List Patterns
const	$\rightarrow$	*integer*	Integer
	$\mid$	`True` $\mid$ `False`	Boolean
	$\mid$	`[ ]`	Nil
	$\mid$	`Error` `(` *string* `)`	Error
binop	$\rightarrow$	`+` $\mid$ `*` $\mid$ `-` $\mid$ `/` $\mid$ `%`	Integer Arithmetic
	$\mid$	`==` $\mid$ `/=` $\mid$ `<` $\mid$ `>` $\mid$ `<=` $\mid$ `>=`	Integer Comparison
	$\mid$	`&&` $\mid$ `||`	Boolean Arithmetic
unop	$\rightarrow$	`-`	Integer Negation
	$\mid$	`^`	Boolean Negation
cdefs	$\rightarrow$	$id_1^{T_1}$ `=` exp_1 ... $id_n^{T_n}$ `=` exp_n	Constant Definitions $(n \geq 0)$
fdefs	$\rightarrow$	$id_1^{T_1}$ `(` $id_{11}^{T_{11}}$ `,` ... `,` $id_{1n_1}^{T_{1n_1}}$ `)` `=` exp_1 ... $id_m^{T_m}$ `(` $id_{m1}^{T_{m1}}$ `,` ... `,` $id_{mn_m}^{T_{mn_m}}$ `)` `=` exp_m	Function Definitions $(m \geq 0, \forall 1 \leq i \leq m : n_i > 0)$

Figure 1: Abstract Syntax

between runtime errors and compile-time errors, since the border is vague and depends on the cleverness of the compiler rather than on the properties of the language.

We will denote by $A_{\underline{\perp}}$ the *doubly lifted domain* $(A_\perp)_{\underline{\perp}}$ with the domain ordering $\underline{\perp} \sqsubset \perp \sqsubset a$ $(\forall a \in A)$. *Fixpoint iteration* over a lifted domain is denoted by the operator "fix" and uses $\perp$ (not $\underline{\perp}$!) as its bottom value.

Abstractions constructed by the lambda operator $\underline{\lambda}$ are always *strict* in their argument, i.e. $(\underline{\lambda} x.e)\perp = \perp$ and $(\underline{\lambda} x.e)\underline{\perp} = \underline{\perp}$.

Furthermore, we define the *choice operator* $\rightarrow$ as

$$e_c \rightarrow e_t; e_f \;:=\; \begin{cases} e_t & \text{if } e_c = \text{ true} \\ e_f & \text{if } e_c = \text{ false} \\ \perp & \text{if } e_c = \perp \\ \underline{\perp} & \text{otherwise} \end{cases}$$

Please note that $\rightarrow$ is strict in its first argument (in particular, $(\underline{\perp} \rightarrow e_t; e_f) = \underline{\perp}$) but *not* in both other arguments, e.g. $(\text{true} \rightarrow e_t; \underline{\perp}) = e_t$. We will also frequently use the shortcut $e_c \rightarrow e_t$ for $e_c \rightarrow e_t; \underline{\perp}$ as sort of a *guard condition* asserting that e_c holds.

For any *disjoint union domain* $A + B = \{(1,a)|a \in A\} \cup \{(2,b)|b \in B)\}$, we define the construction operators

$$\begin{array}{l} \text{mkA} : A \rightarrow A + B,\; a \mapsto (1, a) \\ \text{mkB} : B \rightarrow A + B,\; b \mapsto (2, b) \end{array}$$

and the *case operator*

$$\begin{array}{l} \text{case } e_c \text{ of} \\ \quad \text{isA}(x) \rightarrow e_1; \\ \quad \text{isB}(y) \rightarrow e_2 \end{array} \;:=\; \begin{cases} e_1[x \leftarrow a] & \text{if } e_c = (1,a) \text{ for some } a \\ e_2[y \leftarrow b] & \text{if } e_c = (2,b) \text{ for some } b \\ \perp & \text{if } e_c = \perp \\ \underline{\perp} & \text{otherwise} \end{cases}$$

These notions naturally generalize to an arbitrary number of component domains. The shortcut "case e_c of isA$(a) \rightarrow e_1$" stands for "case e_c of isA$(a) \rightarrow e_1$; isB$(b) \rightarrow \underline{\perp}$". Please note that the case operator is only strict in its first argument.

2.3 Semantic Algebras

In the following subsections, we will define several valuation functions:

P :	Program $\rightarrow$ Value	Program Value
E :	Expression $\rightarrow$ ValueEnv $\rightarrow$ Value	Expression Value
T :	Expression $\rightarrow$ TypeEnv $\rightarrow$ Type	Expression Type
DE :	Definitions $\rightarrow$ ValueEnv $\rightarrow$ ValueEnv	Value Environment
DT :	Definitions $\rightarrow$ TypeEnv $\rightarrow$ TypeEnv	Type Environment

For each expression $[\![exp]\!]$, we will define a rule $\mathbf{T}[\![exp]\!]e$ denoting the *(static) type* of $[\![exp]\!]$ and a rule $\mathbf{E}[\![exp]\!]$ denoting the *(dynamic) value* of $[\![exp]\!]$[1].

The domain "Value" is defined as follows:

Bool =	$\{\text{false}, \text{true}\}$	Boolean
Int =	$\{\text{minint}, \ldots, 0, \ldots, \text{maxint}\}$	Integer
Nil =	*Unit*	Empty List
List =	Value $\times$ Value	Non-Empty List
Tuple$_i$ =	Value^i	i-Tuple $(i \geq 2)$
Function$_i$ =	Value^0 $\rightarrow \ldots \rightarrow$ Value^i	i-ary Function $(i \geq 1)$
Value =	(Bool + Int + Nil + List + Tuple$_i$ $(\forall i \geq 2)$ + Function$_i$ $(\forall i \geq 1))_{\underline{\perp}}$	Denotable Value

[1] The terms "static" and "dynamic" refer to the fact, that the definition of **T** copes without fixpoint iteration, while the definition of **E** does not.

The usual logical and arithmetic operations are defined on the domains "Bool' and "Int". The domain *Unit* consists of a single element () (and thus does not carry any information contents). Please note that "Value" is a lifted domain that as well contains "error" as "non-termination".

The domain "Type" is defined by

ErrorType =	*Unit*	Error Value
BoolType =	*Unit*	Boolean
IntType =	*Unit*	Integer
NilType =	*Unit*	Empty List
ListType =	Type	Non-Empty List
TupleType_i =	Type^i	i-Tuples ($i \geq 2$)
FunctionType_i =	Type^{i+1}	i-ary Function ($i \geq 1$)
Type =	(ErrorType + BoolType + IntType + NilType + ListType + TupleType_i ($\forall i \geq 2$) + FunctionType_i ($\forall i \geq 1$)$)_{\underline{\perp}}$	Denotable Type

"Type" is a lifted domain that contains $\underline{\perp}$ to denote "typing error" ($\perp$ is not used). There is no domain corresponding to user-named types, since we assume that these type names have been already replaced by their definition. In the following subsection, we will define several functions operating on "Type".

The environment domains "ValueEnv" and "TypeEnv" are constructed as

ValueEnv =	(Identifier $\rightarrow$ Value$)_{\underline{\perp}}$	Value Environment
TypeEnv =	(Identifier $\rightarrow$ Type$)_{\underline{\perp}}$	Type Environment

We use the symbol [ ] to denote an empty environment that maps every identifier to $\underline{\perp}$. The environment $[[\![id]\!] \leftarrow x]$ maps $[\![id]\!]$ to x and every other identifier to $\underline{\perp}$. $[[\![id]\!] \leftarrow x]\ e$ extends the environment e by a new binding for $[\![id]\!]$ respectively replaces any existing binding for $[\![id]\!]$. Please note that all binding operations are *strict* in their (type respectively value) argument e.g. $[[\![id]\!] \leftarrow \perp]\ e = \perp$.

The following valuation functions will be not refined any more:

I :	Numeral $\rightarrow$ Integer	Integers
FV :	Expression $\rightarrow$ IdentSet	Free Variables
FF :	Expression $\rightarrow$ IdentSet	Free Functions

$\mathbf{I}[\![integer]\!]$ translates the textual description of an integer number into its *numerical value*. The result of the function is $\underline{\perp}$ if this value is outside the range $\{minint, \ldots, maxint\}$.

$\mathbf{FV}[\![exp]\!]$ is the set of *free variables* in exp i.e. the identifiers of those variables that are referenced but not defined within exp. Likewise, $\mathbf{FF}[\![exp]\!]$ denotes the set of *free functions* in exp i.e. the identifiers of those functions that are applied but not defined within exp.

2.4 Typing

We assume that user-defined type names have been replaced by their definitions and that polymorphic functions have been replaced by their monomorphic counterparts. Thus on the level of the core language types are constructed from a small set of type constants and constructors.

However even on the core level "type compatibility" still does not mean "type equality". The reason is that two of the types need special consideration:

- "mkErrorType()" is compatible with any type.
- "mkNilType()" is compatible with any list type.

These types are not available at the user-level of the language but arise from the polymorphic constants `Error` and `[ ]` without which programming would be rather cumbersome.

The following function defines when two types are considered compatible by the denotation function **T**:

$$
\begin{array}{l}
\text{eqType} : \text{Type} \times \text{Type} \rightarrow \text{Bool} \\
\text{eqType}(T_1, T_2) = \\
\quad T_1 = \text{ mkErrorType}() \vee T_2 = \text{ mkErrorType}() \rightarrow \text{ true}; \\
\quad \text{case } T_1 \text{ of} \\
\quad\quad \text{isIntType}() \rightarrow T_2 = \text{ mkIntType}() \\
\quad\quad \text{isBoolType}() \rightarrow T_2 = \text{ mkBoolType}() \\
\quad\quad \text{isTupleType}_n(t_1, \ldots, t_n) \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isTupleType}_m(s_1, \ldots, s_m) \rightarrow \\
\quad\quad\quad\quad\quad m = n \wedge \forall 1 \leq i \leq m : \text{ eqType}(t_i, s_i) \\
\quad\quad\quad\quad \text{otherwise } \rightarrow \text{ false} \\
\quad\quad \text{isNilType}() \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isNilType}() \rightarrow \text{ true} \\
\quad\quad\quad\quad \text{isListType}(t) \rightarrow \text{ true} \\
\quad\quad\quad\quad \text{otherwise } \rightarrow \text{ false} \\
\quad\quad \text{isListType}(t_1) \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isNilType}() \rightarrow \text{ true} \\
\quad\quad\quad\quad \text{isListType}(t_2) \rightarrow \text{ eqType}(t_1, t_2) \\
\quad\quad\quad\quad \text{otherwise } \rightarrow \text{ false}
\end{array}
$$

The following function returns the more defined of two compatible types. This is required to determine the type of "multi-branched" expressions such as conditional or case expressions.

$$
\begin{array}{l}
\text{joinType} : \text{Type} \times \text{Type} \rightarrow \text{Type} \\
\text{joinType}(T_1, T_2) = \\
\quad T_1 = \text{ mkErrorType}() \rightarrow T_2; \\
\quad T_2 = \text{ mkErrorType}() \rightarrow T_1; \\
\quad \text{case } T_1 \text{ of} \\
\quad\quad \text{isIntType}() \rightarrow (T_2 = \text{ mkIntType}() \rightarrow T_1) \\
\quad\quad \text{isBoolType}() \rightarrow (T_2 = \text{ mkBoolType}() \rightarrow T_1) \\
\quad\quad \text{isTupleType}_n(t_1, \ldots, t_n) \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of isTupleType}_m(s_1, \ldots, s_m) \rightarrow \\
\quad\quad\quad\quad m = n \rightarrow \text{ mkTupleType(joinType}(t_1, s_1), \ldots, \text{joinType}(t_n, s_n)) \\
\quad\quad \text{isNilType}() \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isNilType}() \rightarrow \text{ mkNilType}() \\
\quad\quad\quad\quad \text{isListType}(t) \rightarrow T_2 \\
\quad\quad \text{isListType}(t_1) \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isNilType}() \rightarrow T_1 \\
\quad\quad\quad\quad \text{isListType}(t_2) \rightarrow \text{ mkListType(joinType}(t_1, t_2))
\end{array}
$$

2.5 Program

$$\mathbf{P}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!] = \mathbf{E}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!][\,]$$

A program is an expression that is evaluated in an empty environment. The denotation function **E** takes as an additional argument the current environment defining the value of each global constant and the definition of each function. **E** is *strict* in this argument since it is defined by rules of the form

$$\mathbf{E}\llbracket exp \rrbracket = \underline{\lambda} e. \ldots$$

(for every kind of expression *exp*) using the strict lambda operator $\underline{\lambda}$. Furthermore, since an environment is strict in its individual bindings, the denotation of an expression will be $\underline{\perp}$ respectively $\perp$ whenever any identifier in its environment is bound to $\underline{\perp}$ respectively $\perp$.

In the following subsections, we will give for each kind of expression a rule for the definition of **E** (the value denotation) and a separate rule for the definition of **T** (the type denotation). The definition of **T** takes an environment as an additional argument that gives the type of each global constant or function.

Each rule of the form $\mathbf{E}\llbracket exp \rrbracket = \underline{\lambda} e.v$ has then to be interpreted as

$$\mathbf{E}\llbracket exp \rrbracket = \underline{\lambda} e.\underline{\lambda} t.\mathbf{T}\llbracket exp \rrbracket t \neq \underline{\perp} \rightarrow v$$

i.e. only well-typed programs have a valid denotation.

2.6 Function Definitions

$$\begin{aligned}
&\mathbf{T}\llbracket \texttt{let}\ fdefs\ \texttt{in}\ exp \rrbracket = \underline{\lambda} e.\mathbf{T}\llbracket exp \rrbracket(\mathbf{DT}\llbracket fdefs \rrbracket e) \\
&\mathbf{E}\llbracket \texttt{let}\ fdefs\ \texttt{in}\ exp \rrbracket = \underline{\lambda} e.\mathbf{E}\llbracket exp \rrbracket(\mathbf{DE}\llbracket fdefs \rrbracket e)
\end{aligned}$$

The type of the return value of a function must match the declared return type:

$$\begin{aligned}
&\mathbf{DT}\llbracket id_1^{T_1}\texttt{(}id_{11}^{T_{11}}\texttt{,} \ldots \texttt{,} id_{1n_1}^{T_{1n_1}}\texttt{)} \texttt{ = } exp_1 \ \ldots\ id_m^{T_m}\texttt{(}id_{m1}^{T_{m1}}\texttt{,} \ldots \texttt{,} id_{mn_m}^{T_{mn_m}}\texttt{)} \texttt{ = } exp_m \rrbracket = \underline{\lambda} e. \\
&\quad (\forall 1 \leq i,j \leq m : i = j \vee \llbracket id_i \rrbracket \neq \llbracket id_j \rrbracket) \rightarrow \\
&\quad\quad \text{let } e' = \\
&\quad\quad\quad [\llbracket id_1 \rrbracket \leftarrow \text{ mkFunctionType}(T_1, T_{11}, \ldots, T_{1n_1}) \\
&\quad\quad\quad \ldots \\
&\quad\quad\quad \llbracket id_m \rrbracket \leftarrow \text{ mkFunctionType}(T_m, T_{m1}, \ldots, T_{mn_m})]\ e \\
&\quad\quad \text{in} \\
&\quad\quad\quad (\text{eqType}(T_1, \mathbf{T}\llbracket exp_1 \rrbracket[\llbracket id_{11} \rrbracket \leftarrow T_{11} \ldots \llbracket id_{1n_1} \rrbracket \leftarrow T_{1n_1}]\ e') \\
&\quad\quad\quad \wedge\ \ldots\ \wedge \\
&\quad\quad\quad \text{eqType}(T_m, \mathbf{T}\llbracket exp_m \rrbracket[\llbracket id_{m1} \rrbracket \leftarrow T_{m1} \ldots \llbracket id_{mn_m} \rrbracket \leftarrow T_{mn_m}]\ e')) \rightarrow e'
\end{aligned}$$

Within a block, the order of function definitions does not matter. Each occurrence of the name of a function defined in this block refers to this particular definition (even if the reference precedes the definition and a function of the same name has been defined in the outer environment). Consequently, function definitions may be (mutually) recursive. It is not allowed to define the same name twice within a block.

$$\begin{aligned}
&\mathbf{DE}\llbracket id_1^{T_1}\texttt{(}id_{11}^{T_{11}}\texttt{,} \ldots \texttt{,} id_{1n_1}^{T_{1n_1}}\texttt{)} \texttt{ = } exp_1 \ \ldots\ id_m^{T_m}\texttt{(}id_{m1}^{T_{m1}}\texttt{,} \ldots \texttt{,} id_{mn_m}^{T_{mn_m}}\texttt{)} \texttt{ = } exp_m \rrbracket = \underline{\lambda} e. \\
&\quad \text{fix}(\lambda e'. \\
&\quad\quad [\llbracket id_1 \rrbracket \leftarrow \text{ mkFunction}_{n_1}(\underline{\lambda} v_1 \ldots \underline{\lambda} v_{n_1}. \\
&\quad\quad\quad \mathbf{E}\llbracket exp_1 \rrbracket[\llbracket id_{11} \rrbracket \leftarrow v_1 \ldots \llbracket id_{1n_1} \rrbracket \leftarrow v_{n_1}]\ e') \\
&\quad\quad \ldots \\
&\quad\quad \llbracket id_m \rrbracket \leftarrow \text{ mkFunction}_{n_m}(\underline{\lambda} v_1 \ldots \underline{\lambda} v_{n_m}. \\
&\quad\quad\quad \mathbf{E}\llbracket exp_m \rrbracket[\llbracket id_{m1} \rrbracket \leftarrow v_1 \ldots \llbracket id_{mn_m} \rrbracket \leftarrow v_{n_m}]\ e')]\ e)
\end{aligned}$$

Functions have at least one argument; instead of a zero-argument function, a constant must be defined.

2.7 Constant Definitions

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{let}\ \mathit{cdefs}\ \texttt{in}\ \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{T}[\![\mathit{exp}]\!](\mathbf{DT}[\![\mathit{cdefs}]\!]e) \\
\mathbf{E}[\![\texttt{let}\ \mathit{cdefs}\ \texttt{in}\ \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{E}[\![\mathit{exp}]\!](\mathbf{DE}[\![\mathit{cdefs}]\!]e)
\end{array}
$$

The type of a constant value must match the declared type. In the full language, it is actually not required to explicitly type constants; the type of a constant is just the type of its value. However, by explicit type annotations, it is possible to give a constant name a type that is more specific than the type of its value (in particular a concrete list type for a constant defined to be the empty list).

$$
\begin{array}{l}
\mathbf{DT}[\![\mathit{id}_1^{T_1} = \mathit{exp}_1\ ;\ \ldots\ ;\ \mathit{id}_n^{T_n} = \mathit{exp}_n]\!] = \underline{\lambda} e. \\
\quad (\forall 1 \leq i, j \leq n : i = j \vee [\![\mathit{id}_i]\!] \neq [\![\mathit{id}_j]\!]) \rightarrow \\
\quad\quad \text{let } e_1 = \ \text{eqType}(T_{p_1}, \mathbf{T}[\![\mathit{exp}_{p_1}]\!]e) \rightarrow [[\![\mathit{id}_{p_1}]\!] \leftarrow T_{p_1}]\ e_0 \text{ in} \\
\quad\quad \text{let } e_2 = \ \text{eqType}(T_{p_2}, \mathbf{T}[\![\mathit{exp}_{p_2}]\!]e_1) \rightarrow [[\![\mathit{id}_{p_2}]\!] \leftarrow T_{p_2}]\ e_1 \text{ in} \\
\quad\quad \ldots \\
\quad\quad \text{let } e_n = \ \text{eqType}(T_{p_n}, \mathbf{T}[\![\mathit{exp}_{p_n}]\!]e_{n-1}) \rightarrow [[\![\mathit{id}_{p_n}]\!] \leftarrow T_{p_n}]\ e_{n-1} \text{ in} \\
\quad\quad\quad e_n
\end{array}
$$

As for function definitions, within a block the order of constant definitions does not matter. Each occurrence of a constant name defined in the block refers to this particular definition. However, in contrast to function definitions, constant definitions must *not* be recursive or mutually recursive. There must exist some reordering of constant definitions in which each reference to a constant is preceded by its definition.

$$
\begin{array}{l}
\mathbf{DE}[\![\mathit{id}_1^{T_1} = \mathit{exp}_1\ ;\ \ldots\ ;\ \mathit{id}_n^{T_n} = \mathit{exp}_n]\!] = \underline{\lambda} e. \\
\quad \text{let } e_1 = [[\![\mathit{id}_{p_1}]\!] \leftarrow \mathbf{E}[\![\mathit{exp}_{p_1}]\!]e_0]\ e \text{ in} \\
\quad \text{let } e_2 = [[\![\mathit{id}_{p_2}]\!] \leftarrow \mathbf{E}[\![\mathit{exp}_{p_2}]\!]e_1]\ e_1 \text{ in} \\
\quad \ldots \\
\quad \text{let } e_n = [[\![\mathit{id}_{p_n}]\!] \leftarrow \mathbf{E}[\![\mathit{exp}_{p_n}]\!]e_{n-1}]\ e_{n-1} \text{ in} \\
\quad\quad e_n
\end{array}
$$

$$
p = \text{topSort}(\{1 \ldots n\}, \{(i,j) | i,j \in \ \{1 \ldots n\}, [\![\mathit{id}_j]\!] \in \mathbf{FV}[\![\mathit{exp}_i]\!]\})
$$

$p = \text{topSort}(V,E)$ is a topological order of the graph (V, E). If such an order does not exist (i.e. the graph contains cycles), the result of topSort (and thus also the environment returned by **DT** respectively **DE**) is $\underline{\perp}$.

2.8 Conditional

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{if}\ \mathit{exp}_c\ \texttt{then}\ \mathit{exp}_t\ \texttt{else}\ \mathit{exp}_f]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![\mathit{exp}_c]\!]e,\ \text{mkBoolType}()) \rightarrow \\
\quad\quad \text{joinType}(\mathbf{T}[\![\mathit{exp}_t]\!]e, \mathbf{T}[\![\mathit{exp}_f]\!]e) \\
\mathbf{E}[\![\texttt{if}\ \mathit{exp}_c\ \texttt{then}\ \mathit{exp}_t\ \texttt{else}\ \mathit{exp}_f]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{E}[\![\mathit{exp}_c]\!]e) \text{ of isBool}(v) \rightarrow \\
\quad\quad v \rightarrow \mathbf{E}[\![\mathit{exp}_t]\!]e; \mathbf{E}[\![\mathit{exp}_f]\!]e
\end{array}
$$

The conditional construct is one of the few operations in the language that is non-strict in some of its arguments (namely the the branch expressions). Consequently, it may encapsulate in one branch a non-terminating or erroneous subcomputation; if the other branch is selected and returns a valid result, the overall result is also valid. However, the types of both branches must be compatible, even if the selection expression always determines the same branch.

2.9 List Selection

$\mathbf{T}[\![\texttt{case}\ exp_s\ \texttt{of [ ] =>}\ exp_n\ \texttt{;}\ id_h\texttt{:}id_t\ \texttt{=>}\ exp_c]\!] = \underline{\lambda} e.$
  case $\mathbf{T}[\![exp_s]\!]e$ of
    isErrorType() $\rightarrow$ joinType($\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e$);
    isNilType() $\rightarrow$ joinType($\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e$);
    isListType(T) $\rightarrow$
      joinType($\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!][[\![id_h]\!] \leftarrow T, [\![id_t]\!] \leftarrow$ mkListType(T)$]\ e$

There is a slight problem in typing the `case` construct if the selection expression is the constant `[ ]` or `Error`. In this pathological case we are not able to derive types for the head and tail identifiers of the second branch; these identifiers must not be used then.

$\mathbf{E}[\![\texttt{case}\ exp_s\ \texttt{of [ ] =>}\ exp_n\ \texttt{;}\ id_h\texttt{:}id_t\ \texttt{=>}\ exp_c]\!] = \underline{\lambda} e.$
  case ($\mathbf{E}[\![exp_s]\!]e$) of
    isNil() $\rightarrow \mathbf{E}[\![exp_n]\!]e$;
    isList(h, t) $\rightarrow\ \mathbf{E}[\![exp_c]\!][[\![id_h]\!] \leftarrow h, [\![id_t]\!] \leftarrow t]\ e$

Like the conditional, the case expression is only strict in its first argument and may encapsulate a non-terminating or erroneous branch expression.

2.10 List Construction

$\mathbf{T}[\![exp_h\ :\ exp_t]\!] = \underline{\lambda} e.$
  case ($\mathbf{T}[\![exp_t]\!]e$) of
    isErrorType() $\rightarrow$ mkListType($\mathbf{T}[\![exp_h]\!]e$)
    isNilType() $\rightarrow$ mkListType($\mathbf{T}[\![exp_h]\!]e$)
    isListType(t) $\rightarrow$ joinType($t, \mathbf{T}[\![exp_h]\!]e$)
$\mathbf{E}[\![exp_h\ :\ exp_t]\!] = \underline{\lambda} e.$
  mkList($\mathbf{E}[\![exp_h]\!]e, \mathbf{E}[\![exp_t]\!]e$)

In constructor expressions, the type of the list head must match the type of the list tail (unless the tail is `[ ]` or `Error`).

2.11 Tuple Selection

$\mathbf{T}[\![exp\ .\ integer]\!] = \underline{\lambda} e.$
  case ($\mathbf{T}[\![exp]\!]e$) of
    isErrorType() $\rightarrow$ isErrorType()
    isTupleType$_n(t_1, \ldots, t_n) \rightarrow$
      let $i = \mathbf{I}[\![integer]\!]$
      in $1 \leq i \leq n \rightarrow t_i$
$\mathbf{E}[\![exp\ .\ integer]\!] = \underline{\lambda} e.$
  case ($\mathbf{E}[\![exp]\!]e$) of isTuple$_n(v_1, \ldots, v_n) \rightarrow$
    let $i = \mathbf{I}[\![integer]\!]$
    in $1 \leq i \leq n \rightarrow v_i$

This form of the selection of a tuple field is actually not available in the full language. It is a compiler-internal translation of the user-level matching construct `(`a_1`, ..., `a_n`) =` t where the a_i become the fields of the tuple t.

2.12 Tuple Construction

$\mathbf{T}[\![(exp_1\texttt{, ... ,}\ exp_n)]\!] = \underline{\lambda} e.$

$$
\begin{aligned}
&\quad \text{mkTupleType}_n(\mathbf{T}[\![exp_1]\!]e, \ldots, \mathbf{T}[\![exp_n]\!]e)\\
&\mathbf{E}[\![\texttt{(}exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)}]\!] = \underline{\lambda} e.\\
&\quad \text{mkTuple}_n(\mathbf{E}[\![exp_1]\!]e, \ldots, \mathbf{E}[\![exp_n]\!]e)
\end{aligned}
$$

In the full language, a single field tuple `(`t`)` is identified with its content t as well as a single field tuple type with its base type. Hence, tuples in the core language have at least two fields.

2.13 Function Application

$$
\begin{aligned}
&\mathbf{T}[\![id\texttt{(}\ exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)}]\!] = \underline{\lambda} e.\\
&\quad \text{case } e([\![id]\!]) \text{ of isFunctionType}_m(t_r, t_1, \ldots\ t_m) \rightarrow\\
&\qquad (m = n \wedge \forall 1 \leq i \leq n :\ \text{eqType}(t_i, \mathbf{T}[\![exp_i]\!]e)) \rightarrow t_r\\
&\mathbf{E}[\![id\texttt{(}\ exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)}]\!] = \underline{\lambda} e.\\
&\quad \text{case } e([\![id]\!]) \text{ of isFunction}(f) \rightarrow\\
&\qquad f\mathbf{E}[\![exp_1]\!]e\ \ \ldots\ \ \mathbf{E}[\![exp_n]\!]e
\end{aligned}
$$

Functions have at least one argument, zero-argument functions are subsumed by constants.

2.14 Binary Expressions

On integers a variety of binary operations are defined. We only give the semantics of two examples; the remaining definitions are similar. Please note that, although the operation symbols themselves determine the result type, the result type is only defined if also the argument types are correct. By this we force **T** to reject constructions like `1 + if True then 2 else [ ]` that are allowed by **E**.

$$
\begin{aligned}
&\mathbf{T}[\![exp_1\ \texttt{+}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{eqType}(\mathbf{T}[\![exp_1]\!]e,\ \text{mkIntType}()) \rightarrow\\
&\quad \text{eqType}(\mathbf{T}[\![exp_2]\!]e,\ \text{mkIntType}()) \rightarrow\\
&\qquad \text{mkIntType}()\\
&\mathbf{E}[\![exp_1\ \texttt{+}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{case } \mathbf{E}[\![exp_1]\!]e \text{ of isInt}(i_1) \rightarrow\\
&\quad \text{case } \mathbf{E}[\![exp_2]\!]e \text{ of isInt}(i_2) \rightarrow\\
&\qquad \text{mkInt}(i_1 + i_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{T}[\![exp_1\ \texttt{==}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{eqType}(\mathbf{T}[\![exp_1]\!]e,\ \text{mkIntType}()) \rightarrow\\
&\quad \text{eqType}(\mathbf{T}[\![exp_2]\!]e,\ \text{mkIntType}()) \rightarrow\\
&\qquad \text{mkBoolType}()\\
&\mathbf{E}[\![exp_1\ \texttt{==}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{case } \mathbf{E}[\![exp_1]\!]e \text{ of isInt}(i_1) \rightarrow\\
&\quad \text{case } \mathbf{E}[\![exp_2]\!]e \text{ of isInt}(i_2) \rightarrow\\
&\qquad \text{mkBool}(i_1 = i_2)
\end{aligned}
$$

Logical conjunction and disjunction are only strict in their first argument:

$$
\begin{aligned}
&\mathbf{T}[\![exp_1\ \texttt{\&\&}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{eqType}(\mathbf{T}[\![exp_1]\!]e,\ \text{mkBoolType}()) \rightarrow\\
&\quad \text{eqType}(\mathbf{T}[\![exp_2]\!]e,\ \text{mkBoolType}()) \rightarrow\\
&\qquad \text{mkBoolType}()\\
&\mathbf{E}[\![exp_1\ \texttt{\&\&}\ exp_2]\!] = \underline{\lambda} e.\\
&\quad \text{case } \mathbf{E}[\![exp_1]\!]e \text{ of isBool}(b_1) \rightarrow b_1 \rightarrow\\
&\qquad \text{case } \mathbf{E}[\![exp_2]\!]e \text{ of isBool}(b_2) \rightarrow \text{mkBool}(b_2);\\
&\qquad \text{mkBool}(\text{false})
\end{aligned}
$$

$$
\begin{array}{l}
\mathbf{T}[\![exp_1 \;\texttt{||}\; exp_2]\!] = \underline{\lambda} e. \\
\quad \mathrm{eqType}(\mathbf{T}[\![exp_1]\!] e,\ \mathrm{mkBoolType}()) \to \\
\quad \mathrm{eqType}(\mathbf{T}[\![exp_2]\!] e,\ \mathrm{mkBoolType}()) \;\to \\
\qquad \mathrm{mkBoolType}() \\
\mathbf{E}[\![exp_1 \;\texttt{||}\; exp_2]\!] = \underline{\lambda} e. \\
\quad \mathrm{case}\ \mathbf{E}[\![exp_1]\!] e\ \mathrm{of}\ \mathrm{isBool}(b_1) \to b_1 \to \\
\qquad \mathrm{mkBool}(\mathrm{true}); \\
\qquad \mathrm{case}\ \mathbf{E}[\![exp_2]\!] e\ \mathrm{of}\ \mathrm{isBool}(b_2) \to \mathrm{mkBool}(b_2)
\end{array}
$$

Actually we have a tight correspondence between the logical operators and the conditional construct, e.g. $\mathbf{E}[\![exp_1 \;\texttt{\&\&}\; exp_2]\!] = \mathbf{E}[\![\texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else False}]\!]$. However, we keep them distinct since the compiler applies different optimizations and generates different target code for both constructs.

2.15 Unary Expressions

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{-}\ exp]\!] = \underline{\lambda} e.\ \mathrm{eqType}(\mathbf{T}[\![exp]\!] e,\ \mathrm{mkIntType}()) \;\to\; \mathrm{mkIntType}() \\
\mathbf{E}[\![\texttt{-}\ exp]\!] = \underline{\lambda} e.\ \mathrm{case}\ \mathbf{E}[\![exp]\!] e\ \mathrm{of}\ \mathrm{isInt}(i) \to \mathrm{mkInt}(-i)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{\^{}}\ exp]\!] = \underline{\lambda} e.\ \mathrm{eqType}(\mathbf{T}[\![exp]\!] e,\ \mathrm{mkBoolType}()) \;\to\; \mathrm{mkBoolType}() \\
\mathbf{E}[\![\texttt{\^{}}\ exp]\!] = \underline{\lambda} e.\ \mathrm{case}\ \mathbf{E}[\![exp]\!] e\ \mathrm{of}\ \mathrm{isBool}(b) \to \mathrm{mkBool}(\neg b)
\end{array}
$$

For type checking, the same remarks apply as in the case of binary expressions.

2.16 Constants

$$
\begin{array}{l}
\mathbf{T}[\![integer]\!] = \underline{\lambda} e.\ \mathrm{mkIntType}() \\
\mathbf{E}[\![integer]\!] = \underline{\lambda} e.\ \mathrm{mkInt}(\mathbf{I}[\![integer]\!])
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{True}]\!] = \underline{\lambda} e.\ \mathrm{mkBoolType}() \\
\mathbf{E}[\![\texttt{True}]\!] = \underline{\lambda} e.\ \mathrm{mkBool}(\mathrm{true})
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{False}]\!] = \underline{\lambda} e.\ \mathrm{mkBoolType}() \\
\mathbf{E}[\![\texttt{False}]\!] = \underline{\lambda} e.\ \mathrm{mkBool}(\mathrm{false})
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{[ ]}]\!] = \underline{\lambda} e.\ \mathrm{mkNilType}() \\
\mathbf{E}[\![\texttt{[ ]}]\!] = \underline{\lambda} e.\ \mathrm{mkNil}()
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{Error (}\ string\ \texttt{)}]\!] = \underline{\lambda} e.\ \mathrm{mkErrorType}() \\
\mathbf{E}[\![\texttt{Error (}\ string\ \texttt{)}]\!] = \underline{\lambda} e. \underline{\perp}
\end{array}
$$

Please note that while the value denotation of `Error` is $\underline{\perp}$, its type denotation is not. The application of $\underline{\perp}$ in the definition of $\mathbf{T}$ denotes a typing error that is propagated to the top. However, the type of an expression involving `Error` is in general different from "mkErrorType()" e.g. `if x <= 0 then Error ("neg") else x-1` is of type "mkIntType()".

2.17 Variables

$$
\begin{array}{l}
\mathbf{T}[\![id]\!] = \underline{\lambda} e.\ e(id) \\
\mathbf{E}[\![id]\!] = \underline{\lambda} e.\ e(id)
\end{array}
$$

The type respectively value denotation of a variable is $\underline{\perp}$ if the variable is not defined in the current environment.

exp	$\rightarrow$	*composer iterator*	Composition
composer	$\rightarrow$	`comp` `(` *id* `,` *exp* `)`	Composer
	$\mid$	`rcomp` `(` *id* `,` *exp* `)`	
iterator	$\rightarrow$	`[` *exp* `|` $iter_1$ `|` ... `|` $iter_n$ `]`	Iterator List
iter	$\rightarrow$	pre_1 `,` ... `,` pre_n `;` *exp* `;` $post_1$ `,` ... `,` $post_m$	Iterator
pre	$\rightarrow$	id_h `:` id_t `<-` *exp*	List Iteration
	$\mid$	*id* `<-` exp_s `..` exp_e `+=` exp_i	Upward Iteration
	$\mid$	*id* `<-` exp_s `..` exp_e `-=` exp_i	Downward Iteration
	$\mid$	*id* `=` *exp*	General Iteration
	$\mid$	*def*	Local Definition
post	$\rightarrow$	*id* `'` `=` *exp*	General Iteration
	$\mid$	*def*	Local Definition

Figure 2: Iterated Composition Syntax

3 Iterated Compositions

3.1 Definition

The core language is basically first-order; however, there exist *iterated compositions* as a sort of higher-order constructs to express typical program patterns without applying recursion. An iterated composition consists of two parts (see Figure 2):

- The *iterator* generates a sequence of variable bindings. An expression is evaluated in each of these environments yielding a sequence of values.
- The *composer* folds the generated value sequence into a single value. There are several composers that may be parameterized by a user-defined function.

First we will describe how value sequences are folded. Let $\langle a_1, a_2, \ldots, a_n \rangle$ be the sequence of values generated by the iterator part of a iterated composition, f the name of a function and b an expression:

- *Standard Composition:* The composer `comp`(f, b) computes the value

  $$f(a_1, f(a_2, \ldots, f(a_n, b)))$$

- *Reverse Composition:* The composer `rcomp`(f, b) computes the value

  $$f(a_n, f(a_{n-1}, \ldots, f(a_1, b), \ldots))$$

`comp` and `compr` essentially correspond to the higher order functions `foldr` (fold right) and `foldl` (fold left) of several functional languages.

The composer function must be of type $f : (A, B) \rightarrow B$ where A is the type of the a_i and B is the type of b which is also the type of the composition result (b, if the value sequence is empty).

There are several other composers that have been predefined in terms of `comp` and `rcomp`:

`list`	=	`comp(:,[ ])`	List Construction
`rlist`	=	`rcomp(:,[ ])`	Reverse List Construction
`loop(`b`)`	=	`rcomp(`$\lambda(x,y).x$`,`b`)`	Loop Iteration

The composer `loop(`b`)` returns the last element of the value sequence.

Now let us turn to the iterator part which consists of an expression and the actual iterator:

`[` *exp* `|` *iterator* `]`

exp is evaluated in each of the bindings generated by *iterator* yielding the value sequence $\langle a_1, a_2, \ldots, a_n\rangle$ to be combined. *iterator* consists of a sequence

$iter_1$ `|` ... `|` $iter_n$

For each environment generated by $iter_1$ `|` ... `|` $iter_{i-1}$, $iter_i$ generates a sequence of environments; the concatenation of all these sequences is the environment sequence of $iter_i$. *exp* is evaluated in the environment sequence of $iter_n$.

Each $iter_i$ may be of one of the following forms:

- *List Iterator:*

  h`:`t `<-` l

  creates a sequence of environments in which h is bound consequently to each element l_i of the list l and t is bound to the corresponding rest list l_{i+1}`:` ... `:`l_n. The qualifier `:`t may be omitted.

- *Arithmetic Iterator:*

  x `<-` a`..`b `+=` c

  creates a sequence of environments in which x is bound consequently to $x_0 = a, x_1 = a + c, x_2 = a + 2 * c, \ldots$ as long as $x_i \leq b$. If the qualifier `+=` c is omitted, $c = 1$ is assumed. If the qualifier `-=` c is used, x is bound to $x_0 = a, x_1 = a-c, x_2 = a-2*c, \ldots$ as long as $x_i \geq b$.

- *General Iterator:*

  y `=` r `;` s_y `;` y`'` `=` t_y

  creates a sequence of environments in which y is bound consequently to $y_0 = r, y_1 = t_{y_0}, y_2 = t_{y_1}, \ldots$ as long as s_{y_i} holds.

- *Superposition:* Iterators may be superposed

  pre_1 `,` ... `,` pre_n `;` *cond* `;` $post_m$ `,` ... `,` $post_m$

  where each pre_i may be an arithmetic iterator, a list iterator or the initialization part of a general iterator. *cond* is a boolean condition and each $post_j$ is the successor part of a general iterator. This iterator superposition creates a sequence of environments where all iterated variables are bound simultaneously to their corresponding value sequences. The environment sequence is terminated when at least one of the value sequences is terminated.

  cond may refer to all iterated variables, not only those defined by general iterators, as well as the successor expressions in the $post_j$. Furthermore, each of the pre_i or $post_j$ may be a local definition. The name defined in a pre_i definition may be referenced in any definition (*not* iterator) $pre_{i'}$, in *cond* and in all $post_j$. The name defined in a $post_j$ definition may be referenced in any $post_{j'}$ successor expression or definition.

Furthermore all names defined in a local *pre* or *post* definition of an iterator $iter_i$ may be referenced in all iterators $iter_j$ with $j > i$. Any of the parts *pre*, *cond* and *post* may be empty.

These rules for the superposition of iterators may look rather complicated but become clear from the translation of composition expressions shown in Subsection 3.3.

3.2 Examples

Some examples will demonstrate the application of iterated compositions:

```
map(f,l) = list [ f(x) | x <- l ]
```

construct a list by applying f to each element of l.

```
map(f,l) = list [ y | x <- l, y = f(x) ]
```

does the same using a local definition.

```
reverse(l) = rlist [ x | x <- l ]
```

returns l with the element order reverted.

```
fromto(a,b) = list [ x | x <- a..b ]
```

returns the integer sequence $[a \dots b]$.

```
pow2(n) = list [ x | x = 1; x <= n; x' = x*2 ]
```

constructs the list $[1, 2, 4, 8, \dots, 2^m]$ where $2^m \leq n < 2^{m+1}$.

```
while(p,l) = list [ x | x <- l; p(x) ]
```

returns the longest initial subsequence of l such that for each element p holds.

```
filter(p,l) = list [ x | x <- l |; p(x) ]
```

returns a copy of l with all elements removed for which p does not hold.

```
pairs(l1,l2) = list [ (x1,x2) | x1 <- l1 | x2 <- l2 ]
```

returns the "cross product" of the lists l_1 and l_2.

```
zip(l1,l2) = list [ (x1, x2) | x1 <- l1, x2 <- l2 ]
```

merges the lists l_1 and l_2 into one.

```
forall(p,l) = comp(and, True) [ p(x) | x <- l ]
```

computes $\forall x \in l . p(x)$.

```
append(l1,l2) = comp(cons, l2) [ x | x <- l1 ]
```

appends the lists l_1 and l_2.

```
flatten(l) = comp(append, [ ]) [ x | x <- l ]
```

flattens l by appending all sublists in l.

```
imult(x,y) = rcomp(iadd, izero) [ idmult(x,yi) | yi <- y ]
```

multiplies two integers x and y represented as lists of digits by multiplying each digit y_i with x and adding the products.

```
split(e,l) = rcomp(choose(e), (Nil, Nil)) [ x | x <- l ]
choose(e,x,(a,b)) = if x < e then (x:a,b) else (a,x:b)
```

splits the list l into two sublists of all elements less than e respectively all elements larger than e (in the reverse order).

```
split(e,l) = loop((Nil, Nil)) [ (a,b) |
  x <- l, a = [ ], b = [ ] ;;
  a' = if x > e then x:a else a,
  b' = if x > e then b else x:b ]
```

does exactly the same.

The example programs in Appendices B and C contain several more complicated applications of iterated compositions.

3.3 Translation

Instead of constructing the denotational semantics of iterated compositions, we will define a translation of these expressions into the core language. The basic idea is that each $iter_i$ is translated into a recursive function that is called with the actual bindings of the iterated variables. A sequence $iter_1$ | ... | $iter_n$ is translated into n nested recursive functions (that are lifted to the top-level of the program as shown in Subsection 4.3).

The translation function **TC** is based on **TI** that is defined by recursion over the number of iterators n. We assume that all sorts of iterators and definitions have been identified and grouped together. Figure 3 shows the translation of standard composition.

F is a new function name that is nowhere else used. Like for l_i, a_j and r_k, the values of $\bar{b}_j, \bar{v}_j$ and $\bar{v}$ are determined in the current environment. $defs_1$, s and $defs_2$ may refer to h_i, t_i, x_j and y_k and are evaluated in the denoted order. If an arithmetic iterator contains `-=` instead of `+=`, `+` is replaced by `-` and `>` by `<`.

The translation of reverse composition shown in Figure 4 differs from standard composition in that the new function F gets an additional parameter to denote the base case of recursion and in the construction of the (tail-)recursive function call.

As an example for the translation of an iterated composition take the `fiter` function defined in the previous subsection. Its translation yields (after some beautifying)

```
filter(p, l) =
  case l of
    [ ] => [ ]
    x:t =>
      if p(x)
        then x : filter(p, t)
        else filter(p, t)
```

The iterated composition consists of two iterator levels. However the inner one is just the test of the condition $p(x)$ but does not introduce any new iterated variable bindings. In this case the compiler does not introduce a new function definition but directly inserts the conditional expression into the outer function.

As an example for the translation of a doubly nested iterated composition take the function `pairs` in the previous subsection. Its translation is

```
pairs(l1, l2) =
  let
    pairs1(l1) =
```

$$\mathbf{TC}[\![\texttt{comp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!] = \mathbf{TI}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TI}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] = [\![f(e,v)]\!]$$

$$\mathbf{TI}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}u_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ t_k\texttt{,}\ defs_2\ \texttt{|}\ R]\!][\![f]\!][\![e]\!][\![v]\!] =$$

```
[[ let
     F(z_1, ..., z_m, x_1, ..., x_n, y_1, ..., y_o) =
       case z_1 of
         [ ] => v̄
         h_1:t_1 =>
           ...
             case z_m of
               [ ] => v̄
               h_m:t_m =>
                 if x_1 > b̄_1 || ... || x_n > b̄_n
                   then v̄
                   else let
                     defs_1
                   in if ^s
                     then v̄
                     else let
                       defs_2
                     in TI[[R]][[f]][[e]]
                         [[F(t_1, ..., t_m, x_1+c̄_1, ..., x_n+c̄_n, r_1, ..., r_o)]]
     b̄_j =b_j ; c̄_j = c_j ; v̄ = v
   in
     F(l_1, ..., l_m, a_1, ..., a_n, u_1, ..., u_o)                            ]]
```

Figure 3: Standard Composition

$$\mathbf{TC}[\![\texttt{rcomp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!] = \mathbf{TJ}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TJ}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] \;=\; [\![f\,\texttt{(}e,v\texttt{)}]\!]$$

$$\mathbf{TJ}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}r_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ u_k\texttt{,}\ defs_2\ \texttt{|}\ R]\!][\![f]\!][\![e]\!][\![v]\!] \;=$$

```
[[ let
     F(z1, ..., zm, x1, ..., xn, y1, ..., yo, w) =
       case z1 of
         [ ] => w
         h1:t1 =>
           ...
             case zm of
               [ ] => w
               hm:tm =>
                 if x1 > b̄1 || ... || xn > b̄n
                   then v̄
                   else let
                     defs1
                   in if ^s
                     then v̄
                     else let
                       defs2
                     in F(t1, ..., tm, x1+c̄1, ..., xn+c̄n, r1, ..., ro,
                         TJ[[R]][[f]][[e]][[v]])
     b̄j =bj; c̄j = cj
   in
     F(l1, ..., lm, a1, ..., an, u1, ..., uo, v)                     ]]
```

Figure 4: Reverse Composition

```
        case l1 of
          [ ] => [ ]
          x1:t1 =>
            let
              pairs2(l2) =
                case l2 of
                  [ ] => pairs1(t1)
                  x2:t2 => (x1,x2) : pairs2(t2)
            in pairs2(l2)
    in pairs1(l1)
```

After lambda-lifting (see Subsection 4.3) and some inlining we get

```
pairs(l1, l2) =
  case l1 of
    [ ] => [ ]
    x1:t1 => pairs2(l2, l2, x1, t1)
pairs2(l, l2, x1, t1) =
  case l of
    [ ] => pairs(t1, l2)
    x2:t2 => (x1,x2) : pairs2(t2, l2, x1, t1)
```

4 Program Transformations

This section describes how we transform a program in the full language into a program in the core language. The full language provides limited higher order capabilities, has a polymorphic type system, and allows nested function definitions. The core language is strictly first-order, monomorphic, and contains combinator definitions only.

There are also several transformations whose purpose is to improve the efficiency of the program (see Subsection 4.5).

4.1 Uncurrying

A *curried* function application provides fewer arguments than the definition actually demands. The result of this application is a function that is specialized with respect to these arguments. E.g. the program

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
in
  map(f(a,b), l)
```

applies the unary function $\lambda x.a + b * x$ to each element of l. The compiler transforms this program into

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
  fxy(z) = f(a,b,z)
in
  map(fxy, l)
```

More formally, we apply the following transformation:

$$\mathbf{UC}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l} \ddots \\ \qquad f(a_1,\ldots,a_{m<n}) \end{array}} \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{lll} \ddots & & \\ & \texttt{let} & \bar{f}(x_{m+1},\ldots,x_n) \;\texttt{=}\; f(a_1,\ldots,a_m,x_{m+1},\ldots,x_n) \\ & \texttt{in} & \bar{f} \end{array}} \end{array}\right]\!\!\right]$$

Since curried function names may only occur as arguments of higher order functions, the definition of the new function $\bar{f}$ may be lifted out of this application.

4.2 Specialization

The language allows to parameterize functions by other functions (see the example in the previous subsection). This form of a higher-order function is treated as a *function template* that is instantiated at compile-time e.g.

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
  fxy(z) = f(a,b,z)
  mapfxy(l) = comp(fxy,[ ]) [ x | x <- l ]
in
  mapfxy(l)
```

The transformation process is described formally by

$$\mathbf{SP}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l} \ddots \\ \qquad f(a_1,\ldots,\ a_{i-1},\ g,\ a_{i+1},\ \ldots,a_n) \end{array}} \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{lll} \ddots & & \\ & \texttt{let} & \bar{f}(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n) \;\texttt{=}\; \mathbf{TS}[\![exp]\!][[\![x_i]\!] \leftarrow [\![g]\!]] \\ & \texttt{in} & \bar{f}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n) \end{array}} \end{array}\right]\!\!\right]$$

TS just replaces all occurrences of x_i by g. After the transformation of all applications, the original higher order function is removed from the program.

4.3 Lambda Lifting

The body of a function definition may contain references to constant names that are defined in the global environment e.g.

```
let
  f(x) = x+a
  a = 1
in f(7)
```

For the compilation it is convenient to have only *combinators* i.e. functions whose bodies are "closed" with respect to their parameters. The result of a combinator only depends on its arguments, thus combinators can be easily manipulated and combined (and evaluated on a remote processor).

In order to transform a function into a combinator, we have to add all global variables referenced in the function body to the parameters of the function and to the arguments of each application i.e.

```
let
  f(x,a) = x+a
in let
  a = 1
in f(7,a)
```

The situation becomes more complicated if a function references another function defined in the same block e.g.

```
let
  f(x) = a+g(x)
  g(y) = if y < 0 then b else f(y-1)
  a = 1
  b = 2
in
  f(7)
```

In general we have to add as arguments all global variables referenced by any function defined in the same block and (directly or indirectly) called by the function under consideration:

```
let
  f(x,a,b) = a+g(x,b)
  g(y,a,b) = if y < 0 then b else f(y-1,a,b)
in let
  a = 1
  b = 2
in
  f(7,a,b)
```

More formally, we define the lambda lifting process by a sequence of transformations to be applied to expressions inside out.

4.3.1 Separation

First we separate function definitions from constant definitions lifting the function definitions to the outer level.

$$\mathbf{LL}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f_i\,(p_{i1}\ldots p_{in_i}) \texttt{ = } \mathit{def}_i \\ & \ldots \\ & c_j \texttt{ = } \mathit{exp}_j \\ & \ldots \\ \texttt{in} & \mathit{exp} \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f_i\,(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i}) \texttt{ = } \mathbf{TL}[\![\mathit{def}_i]\!][\ldots f_i \leftarrow q_i \ldots] \\ & \ldots \\ \texttt{in} & \begin{array}{ll} \texttt{let} & \ldots \\ & c_j \texttt{ = } \mathbf{TL}[\![\mathit{exp}_j]\!][\ldots f_i \leftarrow q_i \ldots] \\ & \ldots \\ \texttt{in} & \mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow q_i \ldots] \end{array} \end{array}\right]\!\!\right]$$

where

$$\begin{array}{l} q = \mathrm{seq}\{[\![c_j]\!] \mid [\![c_j]\!] \in \mathbf{FV}[\![\mathit{def}_i]\!] \cup \{\mathbf{FV}[\![\mathit{def}_k]\!] \mid [\![\mathit{def}_i]\!] \rightarrow^*_{FF} [\![\mathit{def}_k]\!]\}\} \\ [\![\mathit{def}_i]\!] \rightarrow_{FF} [\![\mathit{def}_j]\!] \Leftrightarrow f_j \in \mathbf{FF}[\![\mathit{def}_i]\!] \end{array}$$

i.e. q is the sequence of free variables to become function arguments, $\rightarrow_{FF}$ is the relation "contains a call of the function with body" and $\rightarrow^*_{FF}$ is the transitive closure of this relation. $\mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow q \ldots]$ appends in $[\![\mathit{exp}]\!]$ the argument list q to each application of f_i. The transformed f_i do not any more reference the c_j defined in the block and may be lifted to the outer level.

4.3.2 Constant Context

Second if a block of lifted function definitions is enclosed in an outer block of constant definitions, we interchange the nesting of blocks by applying the transformation

$$\mathbf{LL}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & c_j \texttt{ = } \ldots \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l} \ddots \\ \qquad \begin{array}{ll} \texttt{let} & \ldots \\ & f_i\,(p_{i1}\ldots p_{in_i}) \texttt{ = } \mathit{def}_i \\ & \ldots \\ \multicolumn{2}{l}{\texttt{in } \mathit{exp}} \end{array} \end{array}} \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f'_i\,(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i}) \texttt{ = } \mathbf{TL}[\![\mathit{def}_i]\!][\ldots f_i \leftarrow (f'_i, q_i) \ldots] \\ & \ldots \\ \texttt{in} & \begin{array}{ll} \texttt{let} & \ldots \\ & c_j \texttt{ = } \ldots \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l} \ddots \\ \qquad \mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow (f'_i, q_i) \ldots] \end{array}} \end{array} \end{array}\right]\!\!\right]$$

In order to resolve name clashes, the lifted version of a function f_i has to be renamed into f'_i. Thus **TL** has to rename function applications as well.

4.3.3 Function Context

Third if a block of function definitions is contained in another function definition, we draw the inner definitions to the level of the outer one (the global "constants" c_i are the parameters of the outer function):

$$
\mathbf{LL}\left\llbracket
\begin{array}{ll}
\texttt{let} & \ldots \\
 & f(\ldots c_j \ldots) = \\
 & \boxed{\begin{array}{lll}
 \ddots & & \\
 & \texttt{let} & \ldots \\
 & & f_i(p_{i1}\ldots p_{in_i}) = \mathit{def}_i \\
 & & \ldots \\
 & \texttt{in}\ \mathit{exp} &
 \end{array}} \\
 & \ldots \\
\texttt{in} \ldots &
\end{array}
\right\rrbracket =
$$

$$
\left\llbracket
\begin{array}{ll}
\texttt{let} & \ldots \\
 & f(c_1 \ldots c_m) = \\
 & \boxed{\begin{array}{ll}
 \ddots & \\
 & \mathbf{TL}\llbracket \mathit{exp} \rrbracket[\ldots f_i \leftarrow (f'_i, q_i) \ldots]
 \end{array}} \\
 & \ldots \\
 & f'_i(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i}) = \mathbf{TL}\llbracket \mathit{def}_i \rrbracket[\ldots f_i \leftarrow (f'_i, q_i) \ldots] \\
 & \ldots \\
\texttt{in} \ldots &
\end{array}
\right\rrbracket
$$

If none of the two previous transformations can be applied any more, all functions have been transformed into combinators that can be collected in a block at the top-level of the program (as demanded in the abstract syntax of the core language).

4.4 Monomorphization

The language allows the definition of polymorphic functions e.g.

```
let
  prod :: ([*a], [*b]) -> [(*a,*b)]
  prod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
in
  prod([1,2,3], [True, False])
```

The compiler transforms each application of a polymorphic function into the application of a monomorphic instance i.e.

```
let
  prod :: ([*a], [*b]) -> [(*a,*b)]
  prod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
  ibprod :: ([Int], [Bool]) -> [(Int,Bool)]
  ibprod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
in
  ibprod([1,2,3], [True, False])
```

The type of the function instance is derived from the types of the given arguments. This is possible by a simple matching algorithm provided that all type variables of the image type (`[(*a,*b)]` in the example above) are among the type variables of the domain type (`([*a], [*b])`).

The only problem arises if the polymorpic constants `[ ]` or `Error` are applied to a polymorphic function definition. E.g. the application

```
prod([ ], [True, False])
```

cannot be typed by the compiler. A solution in such a (rather rare) case is to define a named constant with a monomorphic type signature:

```
let
  inil :: [Int]
  inil = [ ]
in
  prod(inil, [True, False])
```

After the transformation of all polymorphic function applications the polymorphic function definition is removed from the program.

4.5 Optimizing Transformations

The compiler performs several program transformations whose purpose is to improve the efficiency of the program.

4.5.1 Function Inlining

The compiler inlines the body of every not (directly or indirectly) recursive function into each application of this function:

$$\mathbf{IL}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(p_1 \ldots p_n) \texttt{ = } def \\ & \ldots \\ \texttt{in} & \boxed{\begin{array}{ll} \ddots & \\ & f(a_1 \ldots a_n) \end{array}} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & \ldots \\ \texttt{in} & \boxed{\begin{array}{lll} \ddots & & \\ & \texttt{let} & p_1 \texttt{ = } a_1 \\ & & \ldots \\ & & p_n \texttt{ = } a_n \\ & \texttt{in} & def \end{array}} \end{array}\right]\!\!\right]$$

Like the function application, the `let` construct is strict in the values a_i, thus the termination properties of the program remain unchanged. After inlining, the function definition is removed from the program.

In the body of a mutually recursive function, the compiler also inlines the call of another function if this function is a member of the current "recursion cycle" but is itself not directly recursive. Consequently, the recursion cycle may collapse into a single recursive function (e.g. if it is of the form $f_1 \to f_2 \to \ldots \to f_n \to f_1$ and each $f_{i \neq 1}$ is neither directly recursive nor member of another cycle).

Function inlining increases the size of the program but reduces the number of function calls and allows several further optimizations to be applied.

4.5.2 Constant Inlining

The compiler inlines the value of a constant definition into a reference to this constant, if it is the only one:

$$\mathbf{IL}\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & c\ \texttt{=}\ exp \\ & \ldots \\ \texttt{in} & \boxed{\begin{array}{l}\ddots \\ \qquad c\end{array}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & \ldots \\ \texttt{in} & \\ & exp\end{array}\right]\!\!\right]$$

Since the constant is only referenced once, there is no danger that the value of the constant is computed more than once. The constant definition is finally removed from the program.

Please note that by this optimization we actually slightly change the denotation of the program. Consider e.g. the transformation

$$\mathbf{IL}\left[\!\!\left[\begin{array}{ll}\multicolumn{2}{l}{\texttt{let a = f(b)}} \\ \texttt{in if} & \texttt{b > 0} \\ & \texttt{then 1} \\ & \texttt{else g(a)}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{ll}\texttt{if} & \texttt{b > 0} \\ & \texttt{then 1} \\ & \texttt{else g(f(b))}\end{array}\right]\!\!\right]$$

If the evaluation of `f(b)` runs into non-termination/error, then the original program runs into non-termination/error, too. However, the transformed program returns 1 for $b > 0$. Actually this would be also the result of the original program when evaluated under a non-strict (e.g. lazy) evaluation strategy. Thus we have

$$\mathbf{P}[\![\mathit{program}]\!] \sqsubseteq \mathbf{C}[\![\mathit{program}]\!] \sqsubseteq \mathbf{Q}[\![\mathit{program}]\!]$$

where **P** is the denotation defined in Section 2, **C** is the semantic actually implemented by the compiler and **Q** is a fictive non-strict interpretation of the language. We therefore should not inline a constant that is referenced in a non-strict context (like a conditional). However, currently the compiler is sloppy and omits the strictness check.

4.5.3 Expression Folding

After the inlining phase, the compiler replaces several forms of compound expressions by equivalent but simpler ones. Expressions amenable to folding are for instance

1. *Arithmetic Expressions*

   $$\mathbf{CF}[\![\mathit{int}_1 \texttt{*} \mathit{int}_2]\!] = [\![\mathbf{I}[\![\mathit{int}_1]\!] * \mathbf{I}[\![\mathit{int}_2]\!]]\!]$$

2. *Logical Expressions*

   $$\mathbf{CF}[\![\mathit{exp}\ \texttt{\&\& True}]\!] = [\![\mathit{exp}]\!]$$

3. *Conditionals*

   $$\mathbf{CF}[\![\texttt{if False then}\ \mathit{exp}_1\ \texttt{else}\ \mathit{exp}_2]\!] = [\![\mathit{exp}_2]\!]$$

4. *List Selections*

   $$\mathbf{CF}[\![\texttt{case}\ \mathit{exp}_h\texttt{:}\mathit{exp}_t\ \texttt{of [ ] =>}\ \mathit{exp}_n\ \texttt{;}\ \mathit{id}_h\texttt{:}\mathit{id}_t\ \texttt{=>}\ \mathit{exp}_c]\!] = [\![\texttt{let}\ \mathit{id}_h\ \texttt{=}\ \mathit{exp}_h\ \texttt{;}\ \mathit{id}_t\ \texttt{=}\ \mathit{exp}_t\ \texttt{in}\ \mathit{exp}_c]\!]$$

5. *Tuple Selections*

   $$\mathbf{CF}[\![\texttt{(}\mathit{exp}_1\texttt{,}\ \ldots\ \texttt{,}\ \mathit{exp}_n\texttt{).}\mathit{int}]\!] = [\![\mathit{exp}_{\mathbf{I}[\![\mathit{int}]\!]}]\!]$$

Several of these transformations change the semantics of the program as described in the previous paragraph. For instance, the transformation

$$\mathbf{CF}[\![\texttt{0 * } exp]\!] = [\![0]\!]$$

is actually only valid under the assumption that the evaluation of *exp* returns a result. Again, we value the practical optimization effect more than the theoretical change in the program semantics.

4.5.4 Condition Drawing

In the code generation phase, several optimizations are applied on sequences of assignment statements. These optimizations may dramatically affect the performance of a program by transforming e.g. a recursive function into a loop. If a conditional interrupts such a sequence, these optimizations may be prohibited.

For this reason we attempt to draw conditional expressions to the outermost level thus increasing the size of unconditionally evaluated expressions:

$$\mathbf{CD}\left[\!\!\left[\ \boxed{\begin{array}{lll} \ddots & & \\ & \texttt{if} & exp_c \\ & & \texttt{then}\ exp_t \\ & & \texttt{else}\ exp_f \end{array}}\ \right]\!\!\right] = \left[\!\!\left[\ \begin{array}{ll} \texttt{if} & exp_c \\ & \texttt{then} \\ & \boxed{\begin{array}{ll} \ddots & \\ & exp_t \end{array}} \\ & \texttt{else} \\ & \boxed{\begin{array}{ll} \ddots & \\ & exp_f \end{array}} \end{array}\ \right]\!\!\right]$$

We only apply this transformation if the context expression does not contain another conditional expression whose test condition encloses the conditional to be lifted. Otherwise we would interchange the nesting of conditionals and the transformation would be applied forever.

Furthermore, the transformation is only valid if the context expression does not contain a block of local definitions whose body expression encloses the conditional. Otherwise such a block might define variables that are referenced in the condition exp_c.

However, if a conditional is enclosed by a local definition block, we can still lift the conditional somewhat further assuming that some of the defined variables are not referenced in exp_c:

$$\mathbf{CD}\left[\!\!\left[\ \begin{array}{lll} \texttt{let} & \ldots & \\ & c_i = def_i & \\ & \ldots & \\ \texttt{in} & \texttt{if} & exp_c \\ & & \texttt{then}\ exp_t \\ & & \texttt{else}\ exp_f \end{array}\ \right]\!\!\right] = \left[\!\!\left[\ \begin{array}{lllll} \texttt{let} & \ldots & & & \\ & c_i = def_i & & & \\ & \ldots & & & \\ \texttt{in} & \texttt{if} & exp_c & & \\ & & \texttt{then} & \texttt{let} & \ldots \\ & & & & c_j = def_j \\ & & & & \ldots \\ & & & \texttt{in} & exp_t \\ & & \texttt{else} & \texttt{let} & \ldots \\ & & & & c_j = def_j \\ & & & & \ldots \\ & & & \texttt{in} & exp_f \end{array}\ \right]\!\!\right]$$

The c_j are those variables that are not referenced in exp_c and that are not used in the definition of any variable referenced in exp_c. Drawing the conditional beyond this block is not possible any more.

4.6 Final Processing

As the last step before code generation we finally transform the program representation from an expression into a set of function definitions:

$$\mathbf{FP}\left[\begin{array}{l}\texttt{let}\ \mathit{fdefs}\\ \texttt{in}\ \mathit{exp}\end{array}\right] = \left[\begin{array}{l}\mathit{fdefs}\\ \texttt{dcmain() = }\mathit{exp}\end{array}\right]$$

The parameter-less main function **`dcmain`** will become the external entry point to the program; for more details on the external interface see Section 6.

5 Code Generation and Optimization

In this section we describe the techniques that we apply for generating efficient C code from programs in the functional core language.

5.1 Basic Code Generation

The compiler has a small and well defined interface to the runtime system that consists of a number of basic types and of the constructor and selector functions for lists and for tuples. The compiler does not care for memory management and garbage collection at all but leaves this task entirely to the runtime system (see Section 6).

After the transformation phase, the program consists of a set of closed function definitions. Each function but the main function is (directly or indirectly) recursive and each local constant is referenced at least twice. Otherwise, the function respectively constant would have been inlined and the definition would have been removed.

We translate each function in the core language into a C function. The main difficulty of this translation arises from the fact that the body of a C function consists of a *statement sequence* while the body of a function in the core language is an *expression*. The compilation of a function body is therefore a mapping from expressions to statement sequences.

This mapping is trivial if we are able to map a functional expression *exp* into a corresponding C expression *cexp* and define the body of the C function just as the statement **`return`**(*cexp*). However, there are several functional expressions that cannot be translated into C expressions but have to be compiled into statement sequences preceding the **`return`** statement:

- *Local Definitions:* A block of local definitions is compiled into a sequence of variable declarations and assignments. This sequentialization is possible since constant definitions have been topologically sorted in the transformation phase. To make the generated C code more compact, variable definitions and assignment statements are combined to initialized variable declarations.

  Depending on the context of the definition block, the last statement in the generated sequence is either a **`return`** (if the result of the block is also the result of the function) or the assignment of the result to a variable (if the result is the value of a named constant). If the result is used anonymously, an auxiliary identifier is introduced to receive the result.

  The following rules show the translation of a definition block that represents the body of a function (where t_f represents the result type of f and t_x represents the type of a constant respectively parameter x)

$$\mathbf{DC}\left[\!\!\left[\begin{array}{l} f\ \texttt{(}\ldots p_i \ldots\texttt{)}\ \texttt{=} \\ \quad\texttt{let} \\ \qquad\ldots \\ \qquad c_j \texttt{ = } exp_j \\ \qquad \ldots \\ \quad \texttt{in} \\ \qquad exp \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t_f\ f\ \texttt{(}\ldots t_{p_i}\ p_i \ldots\texttt{)}\ \texttt{\{} \\ \quad \ldots \\ \quad t_{c_j}\ c_j \texttt{ = } exp_j\texttt{;} \\ \quad\ldots \\ \quad \texttt{return(}exp\texttt{);} \\ \texttt{\}} \end{array}\right]\!\!\right]$$

and the translation of a function block whose result is used anonymously.

$$\mathbf{DC}\left[\!\!\left[\boxed{\begin{array}{l} \ddots \\ \qquad\texttt{let} \\ \qquad\quad\ldots \\ \qquad\quad {c_j}^{t_{c_j}} = exp_j \\ \qquad\quad \ldots \\ \qquad \texttt{in} \\ \qquad\quad exp \end{array}}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \ldots \\ t_{c_j}\ c_j \texttt{ = } exp_j\texttt{;} \\ \ldots \\ t_{exp}\ x \texttt{ = } exp\texttt{;} \\ \boxed{\begin{array}{l} \ddots \\ \qquad\quad x \end{array}} \end{array}\right]\!\!\right]$$

During the code generation the context of an expression is propagated to the base expression of a block i.e. if `let` ...`in` *exp* is compiled in the context "function body", then all statement sequences generated from *exp* will be terminated by a **`return`**.

- *Conditional Expressions:* A conditional expression is compiled to the **`if`** statement of the C language. C also offers conditional expressions of the syntactic form c `?` t `:` f. However, we do not want to rely on this feature since it is not available in many other imperative languages and actually not typical for C programs written by humans.

  Each branch of the conditional expression is translated into a statement sequence propagating the context of the conditional expression: if the conditional expression is the body of the function, both generated statement sequences are terminated by a **`return`**. Otherwise, both branches finally assign their result to a (possibly auxiliary) variable.

  We show the translation of a conditional function body

$$\mathbf{DC}\left[\!\!\left[\begin{array}{l} f\ \texttt{(}\ldots p_i \ldots\texttt{)}\ \texttt{=} \\ \quad\texttt{if}\ c \\ \qquad\texttt{then}\ \ldots \\ \qquad\texttt{else}\ \ldots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t_f\ f\ \texttt{(}\ldots t_{p_i}\ p_i \ldots\texttt{)}\ \texttt{\{} \\ \quad \texttt{if (}c\texttt{) \{} \\ \qquad \ldots \\ \qquad \texttt{return(}\ldots\texttt{);} \\ \quad \texttt{\}} \\ \quad \texttt{else \{} \\ \qquad \ldots \\ \qquad \texttt{return(}\ldots\texttt{);} \\ \quad \texttt{\}} \\ \texttt{\}} \end{array}\right]\!\!\right]$$

and of a conditional expression that is used anonymously:

$$\mathbf{DC}\left[\!\!\left[\boxed{\begin{array}{l}\ddots\\ \quad\texttt{if } c\\ \quad\quad\texttt{then } \ldots\\ \quad\quad\texttt{else } \ldots\end{array}}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t\ x;\\ \ldots\\ \texttt{if } (c)\ \{\\ \quad\ldots\\ \quad x = \ldots;\\ \}\\ \texttt{else } \{\\ \quad \ldots\\ \quad x=\ldots;\\ \}\\ \boxed{\begin{array}{l}\ddots\\ \quad\quad x\end{array}}\end{array}\right]\!\!\right]$$

- *Error Expressions:* Any strict expression containing the constant `Error(`*string*`)` is translated into a single statement `ERROR(`*string*`)` that aborts the execution of a program. To make sense such expressions must thus be encapsulated by conditional expressions:

$$\mathbf{DC}\left[\!\!\left[\begin{array}{l}\texttt{if } c\\ \quad\texttt{then } \ldots\\ \quad\texttt{else } \ldots\texttt{Error(}string\texttt{)}\ldots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t\ x;\\ \texttt{if } (c)\ \{\\ \quad\ldots\\ \quad x = \ldots;\\ \}\\ \texttt{else}\\ \quad\texttt{ERROR(}string\texttt{)}\end{array}\right]\!\!\right]$$

The result of the basic code generation phase is a program in the C language consisting of a set of recursive function definitions. The remaining sections discuss how to improve the time and memory efficiency of the program by transforming recursion into iteration.

5.2 Tail-Recursion

Recursion is a rather expensive form of expressing repetitive computations since a recursive function consumes $O(n)$ memory where n is the depth of the call tree. A key strategy to improve the efficiency of a functional program is therefore to transform a recursive function into an iterative loop that copes with $O(1)$ memory and is also considerably faster. In particular, there exists the class of *tail-recursive* functions which can be easily transformed into loops in an automatic fashion.

A tail-recursive function call is a recursive function call whose result is directly forwarded to the caller of the function. E.g. in the function

```
f(x, y) =
  if x == 0
    then y
    else if even(x)
           then 1+f(x-1,y-1)
           else f(x-1,y+1)
```

the call `f(x-1,y+1)` is tail-recursive. A function is called tail-recursive, if all of its recursive calls are tail-recursive.

Above function is compiled into the following C function:

```
int f(int x, int y) {
  if (x == 0)
    return(y);
  else if (even(x))
    return(1+f(x-1,y-1));
  else
    return(f(x-1,y+1));
}
```

The tail-recursive call in this function can be translated into the iteration of a while loop

```
int f(int x, int y) {
  while (TRUE) {
    if (x == 0)
      return(y);
    else if (even(x))
      return(1+f(x-1,y-1));
    else {
      x = x-1;
      y = y+1;
    }
  }
}
```

by destructively updating of function parameters by the arguments of the call. The current function activation performs the job of the tail-recursive one by a straight jump to the begin of the function body.

A tail-recursive function can therefore be transformed into a function that is not recursive at all. However, even if there remain some recursive function calls that are not tail-recursive (as in above example), the depth of the call tree may be considerably decreased.

The jump to the begin of the function can be implemented by a `goto` statement but we prefer the more structured form of a `while` loop. After some beautifying we get the form

```
int f(int x, int y) {
  while (x != 0) {
    if (even(x))
      return(1+f(x-1,y-1));
    else {
      x = x-1;
      y = y+1;
    }
  }
  return(y);
}
```

that a human programmers would actually write.

The transformation is therefore fairly easy, the only problem arises in determining the order in which the parameters are to be updated for the next iteration of a loop. A tail-recursive function call

```
f(x, y) {
  ...
  f(y, x-1)
  ...
}
```

must be compiled into

```
f(x, y) {
  while (TRUE) {
    ...
    z = x
    x = y
    y = z-1
    ...
  }
}
```

where `z` is an auxiliary variable that has to be introduced since there exists a dependency from `x` to `y` and vice versa. More formally, we apply the transformation

TR ⟦
```
t_f f(t_1 x_1, ... , t_n x_n) {
  ...
  [ return(f(a_1, ... , a_n)); ]
  ...
}
```
⟧ = ⟦
```
t_f f(t_1 x_1, ... , t_n x_n) {
  while (TRUE) {
    ...
    [ x_p1 = a_p1;
      ...
      x_pn = a_pn; ]
    ...
  }
}
```
⟧

where p is a topological order of the graph

$$(V, E) = (\{x_1, \ldots, x_n\}, \{(x_i, x_j) | i \neq j \wedge x_i \rightarrow_R x_j\}).$$

$x_i \rightarrow_R x_j$ stands for "x_j is used in the new value for x_i". If such an order does not exist, some dependency must be removed by adding a new variable z to V and replacing each dependence $x_i \rightarrow x_j$ in E by $x_i \rightarrow z$. In other words, an assignment z = x_j is added and all further assignments use z instead of x_j.

After the introduction of a loop we perform a "beautifying transformation"

BT ⟦
```
t_f f(t_1 x_1, ... , t_n x_n) {
  while (TRUE) {
    if (cond) {
      [ ...
        return(exp); ]
    }
    else {
      ...
    }
  }
}
```
⟧ = ⟦
```
t_f f(t_1 x_1, ... , t_n x_n) {
  while (!cond) {
    ...
  }
  [ ...
    return(exp); ]
}
```
⟧

if possible.

The transformation of tail-recursive functions into loops is a standard technique in compilation of functional languages. However, many interesting functions are not tail-recursive. In the following subsections we describe two further optimization techniques that transform frequently occurring forms of recursion into tail-recursion. We are not aware of other compilers applying these forms of optimizations.

5.3 List Results

Consider the core-language function

```
f(x) =
  if x == 0
    then [ ]
    else g(x):f(x-1)
```

which is the general pattern for a large class of list-generating functions; e.g. it is generated from the iterated composition

```
list [ g(x) | x <- n..1 -= 1 ]
```

This function is compiled into the C function

```
intlist f(int x) {
  if (x == 0)
    return(NIL);
  else
    return(CONS(g(x), f(x-1)));
}
```

Unfortunately, the function is not tail-recursive and therefore not amenable to the transformation described in the previous section.

Of course the result list of above function might be constructed also tail-recursively in the reverse order and the tail-recursive function `reverse` might be applied to create the actual result. However this solution is inelegant and requires twice as much time and memory[2].

An experienced programmer will instead write the following function

```
intlist f(int x) {
  intlist y;
  intlist *z = &y;
  while (x != 0) {
    *z = NEW();
    (*z)->head = g(x);
    z = &(*z)->tail;
    x = x-1
  }
  *z = NIL;
  return(y);
}
```

that has the same input/output behaviour but copes with a simple while loop. The main idea is to remember in a variable `z` the address of the tail of the previous list cell which is updated with the pointer to next cell. The function `NEW` allocates such a cell.

In order to automatize this tranformation we introduce the operation

```
cell *RCONS(word x, cell *z) {
  *z = NEW();
  (*z)->head = x;
  return(&(*z)->tail);
}
```

Compare this function with the code above and also with the definition of the `CONS` operation for the construction of a new list cell:

[2] Actually several functions of the SACLIB system are implemented in this style.

```
cell CONS(word x, cell y) {
  cell z = NEW();
  z->head = x;
  z->tail = y;
  return(z);
}
```

The compiler now applies `RCONS` to transform the function `f` at the begin of this subsection into the tail-recursive form

```
void _f(int x, intlist *y) {
  if (x == 0)
    *y = NIL;
  else
    _f(x-1, RCONS(g(x), y));
}
intlist f(int x) {
  intlist y;
  _f(x, &y);
  return(y);
}
```

introducing a procedure `_f` with a tail-recursive call. After transforming this call into a loop we get

```
void _f(int x, intlist *y) {
  while (x != 0) {
    y = RCONS(g(x), y);
    x = x-1;
  }
  *y = NIL;
}
intlist f(int x) {
  intlist y;
  _f(x, &y);
  return(y);
}
```

which (after inlining `_f` and `RCONS`) is exactly the efficient program shown above.

More formally, the compiler applies the following transformation

$$\mathbf{LR}\ \left[\!\!\left[\begin{array}{l} t_f\ f\texttt{(}t_1\ x_1\texttt{,}\ \ldots\ \texttt{,}\ t_n\ x_n\texttt{)}\ \texttt{\{} \\ \quad\ldots \\ \quad\boxed{\texttt{return(}b\texttt{);}} \\ \quad\ldots \\ \quad\boxed{\texttt{return(CONS(}e\texttt{,}\ f\texttt{(}a_1\texttt{,}\ \ldots\ \texttt{,}\ a_n\texttt{)));}} \\ \quad\ldots \\ \texttt{\}} \end{array}\right]\!\!\right] =$$

```
void g(t1 x1, ... , tn xn, tf *y) {
  ...
  *y = b;
  return;
  ...
  g(a1, ... , an, RCONS(e, y));
  return;
  ...
}

tf f(t1 x1, ... , tn xn) {
  tf y;
  g(x1, ... , xn, &y);
  return(y);
}
```

The compiler applies the transformation whenever all recursive calls of a list-valued function are contained in a `return` call of above form.

5.4 Tuple Results

Take a look at the core language function

```
split(e, l) =
  case l of
    Nil => (Nil, Nil)
    h:t => let (a,b) = split(e, t)
           in if h < e
             then (h:a, b)
             else (a, h:b)
```

which splits a list into two result lists. The target code corresponding to this function is

```
listpair split(elem e, list l) {
  if (ISNIL(l))
    return(TUPLE2(NIL, NIL));
  else {
    elem h = HEAD(l);
    if (h < e) {
      listpair p = split(e, t);
      return(TUPLE2(CONS(h, ELEM1(p)), ELEM2(p)));
    }
    else {
      listpair p = split(e, t);
      return(TUPLE2(ELEM1(p), CONS(h, ELEM2(p))));
    }
  }
}
```

Unfortunately the transformation of the previous subsection is not directly applicable to this function, since its result is a tuple object (however containing two lists). Instead the compiler transforms this function into

```
void split2(elem e, list l, list *l1, list *l2) {
  if (ISNIL(l)) {
```

```
      *l1 = NIL; *l2 = NIL;
    }
    else {
      elem h = HEAD(l);
      if (h < e) {
        list la, lb;
        split(e, TAIL(l), &la, &lb);
        *l1 = CONS(h, la); *l2 = lb;
      }
      else {
        list la, lb;
        split(e, TAIL(l), &la, &lb);
        *l1 = la; *l2 = CONS(h, lb);
      }
    }
  }
  listpair split(elem e, list l) {
    list l1, l2;
    split2(e, l, &l1, &l2);
    return(TUPLE2(l1, l2));
  }
```

creating an auxiliary function that takes a reference argument for each field of the result tuple of the original function. Now a peephole optimizer generates

```
  void split2(elem e, list l, list *l1, list *l2) {
    if (ISNIL(l)) {
      *l1 = NIL; *l2 = NIL;
    }
    else {
      elem h = HEAD(l);
      if (h < e)
        split(e, TAIL(l), RCONS(h, l1), l2);
      else
        split(e, TAIL(l), l1, RCONS(h, l2));
    }
  }
```

which is amenable to tail-recursion removal. The final target code thus is

```
  void split2(elem e, list l, list *l1, list *l2) {
    while (!ISNIL(l)) {
      elem h = HEAD(l);
      if (h < e) {
        l = TAIL(l);
        l1 = RCONS(h, l1);
      }
      else {
        l = TAIL(l);
        l2 = RCONS(h, l2);
      }
    }
    *l1 = NIL; *l2 = NIL;
  }
```

which is as efficient as can be reasonably expected.

More formally, we apply the following transformation on a function f generating a tuple with component types $(s_1, \ldots, s_m)$:

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l}
t_f\ f(t_1\ x_1\texttt{,}\ \ldots\ \texttt{,}\ t_n\ x_n\texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad\boxed{\ldots\ f\texttt{(}a_1\texttt{,}\ \ldots\ \texttt{,}\ a_n\texttt{)}\ldots}\\
\quad\ldots\\
\quad\boxed{\texttt{return(}b\texttt{);}}\\
\quad\ldots\\
\texttt{\}}
\end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{l}
\texttt{void}\ g(t_1\ x_1\texttt{,}\ \ldots\ \texttt{,}\ t_n\ x_n\texttt{,}\ s_1\ \texttt{*}y_1\texttt{,}\ \ldots\ \texttt{,}\ s_m\ \texttt{*}y_m\texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad s_1\ z_1\texttt{;}\ \ldots\ \texttt{;}\ s_m\ z_m\texttt{;}\\
\quad\ldots\\
\quad g\texttt{(}a_1\texttt{,}\ \ldots\ \texttt{,}\ a_n\texttt{,}\ \texttt{\&}z_1\texttt{,}\ \ldots\ \texttt{,}\ \texttt{\&}z_m\texttt{);}\\
\quad\ldots\\
\quad\boxed{\ldots\ \texttt{TUPLE}_m\texttt{(}z_1\texttt{,}\ \ldots\ \texttt{,}\ z_m\texttt{)}\ \ldots}\\
\quad\ldots\\
\quad\boxed{\begin{array}{l}\texttt{*}y_1\ \texttt{=}\ b\texttt{.1;}\\ \ldots\\ \texttt{*}y_m\ \texttt{=}\ b\texttt{.m;}\\ \texttt{return;}\end{array}}\\
\quad\ldots\\
\texttt{\}}\\
\\
t_f\ f(t_1\ x_1\texttt{,}\ \ldots\ \texttt{,}\ t_n\ x_n\texttt{)}\ \texttt{\{}\\
\quad s_1\ y_1\texttt{;}\ \ldots\ \texttt{;}\ s_m\ y_m\texttt{;}\\
\quad g\texttt{(}x_1\texttt{,}\ \ldots\ \texttt{,}\ x_n\texttt{,}\ \texttt{\&}y_1\texttt{,}\ \ldots\ \texttt{,}\ \texttt{\&}y_m\texttt{);}\\
\quad\texttt{return(TUPLE}_m(y_1\texttt{,}\ \ldots\ \texttt{,}\ y_m\texttt{));}\\
\texttt{\}}
\end{array}\right]\!\!\right]$$

After this transformation, we apply the peephole optimizers

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l}
\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad t_i\ a_i\texttt{;}\\
\quad\ldots\\
\quad f\texttt{(}\ \ldots\ \texttt{\&}a_i\ \ldots\ \texttt{);}\\
\quad\ldots\\
\quad\texttt{*}x_i\ \texttt{=}\ a_i\texttt{;}\\
\quad\ldots\\
\texttt{\}}
\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}
\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad\ldots\\
\quad f\texttt{(}\ \ldots\ x_i\ \ldots\ \texttt{);}\\
\quad\ldots\\
\quad\ldots\\
\texttt{\}}
\end{array}\right]\!\!\right]$$

and

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l}
\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad t_i\ a_i\texttt{;}\\
\quad\ldots\\
\quad f\texttt{(}\ \ldots\ \texttt{\&}a_i\ \ldots\ \texttt{);}\\
\quad\ldots\\
\quad\texttt{*}x_i\ \texttt{= CONS(}exp\texttt{,}\ a_i\texttt{);}\\
\quad\ldots\\
\texttt{\}}
\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}
\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{)}\ \texttt{\{}\\
\quad\ldots\\
\quad\ldots\\
\quad f\texttt{(}\ \ldots\ \texttt{RCONS(}exp\texttt{,}\ x_i\texttt{)}\ \ldots\ \texttt{);}\\
\quad\ldots\\
\quad\ldots\\
\texttt{\}}
\end{array}\right]\!\!\right]$$

If by these transformations the recursive call of f becomes tail-recursive, it can be transformed into a loop.

Above transformations are applied on all tuple-generating functions. In the program each call of the original function f is replaced by the call of the corresponding new function g (as shown in the new definition of f).

A further peephole optimization step is applied to replace each application `ELEM`$_i$`(`z`)` (where z is any variable holding the value `TUPLE`$_m$`(`z_1`, ... , `z_m`))` by the variable z_i. If all applications of z are of this form, then its definition is removed and the tuple is never generated.

Thus there are only few cases when tuples are actually treated as data structures:

1. A tuple is stored in a data structure.
2. A tuple is returned as the result of an external function.
3. A tuple is passed as a function argument.

Concerning the second case, there is a special form of external function declarations for functions that do not return a tuple but actually expect multiple references to store the function results in (see Section 6).

Concerning the third case, we assume that if the programmer explicitly passes a tuple instead of multiple function arguments he actually expects a data structure to be created. However there is one exception to this rule. In an iterated composition

`rcomp(`f`, `b`) [ `e` | `$iters$` ]`

the base value b becomes an argument of the automatically generated recursive function. Since the only way for a programmer to pass multiple values is to use a tuple for b, the compiler transforms a tuple in this place into multiple function arguments.

6 The External Interface

In this section, we describe the interaction between the code generated by the compiler and external C types, values, and functions.

6.1 The Runtime System

The compiler does not care for the low-level details of data representation and memory management (i.e. garbage collection). The generated code assumes the following types and operations to be implemented by the runtime system:

- *Basic Type:* The atomic type `WORD` is able to hold integers, list references, and tuple references. All types in the D program are mapped to this type in the C program.
- *List Operations:* The functions `CONS`, `RCONS`, `ISNIL`, `HEAD` and `TAIL` are used for list manipulation.
- *Tuple Operations:* The tuple constructors `TUPLE`i, $2 \leq i \leq 6$, and the tuple selectors `ELEM`i, $1 \leq i \leq 6$, manipulate small tuples. For $i > 6$, the list to tuple converter `TUPLE` and the tuple selector `ELEM` are used.

The SACLIB implementation of this interface represents all compound objects as lists; all constructors are boiled down to the `COMP` operation and all selectors down to the functions `FIRST` and `RED`.

Appendix D shows the SACLIB implementation of this interface in detail.

decls	$\rightarrow$	`extern type` *id*	External Type
	|	`extern` *id* `::` *type*	External Argument
	|	`extern const` *id* `::` *type*	External Constant
	|	`extern` *id* `::` *funtype*	External Function
	|	`extern proc` *id* `::` *funtype*	External Procedure

Figure 5: External Declarations

6.2 The Main Function

The compiler translates an expression in the D language into a file that contains the definition of a single external C function (named `dcmain` by default) which returns the value of the expression. While this file may contain the definitions of other C functions as well, these functions may be only called by `dcmain` or other local functions.

Actually, all generated functions but `dcmain` are declared `static` and are thus only visible within the file. Even without this restriction, there is no guarantee that a function contained in the original D program has a corresponding C counterpart (due to renaming, program transformations and in particular inlining).

In short, `dcmain` is the only function that may be called from other object files. Up to now we have assumed that the functional program is a closed expression without any free variables. In this case the main function does not expect any arguments. The following subsection describes how to make this interface more flexible.

6.3 External Types

The compiler allows type names to be declared external. In this case, only the name of the type but not its definition is known to the compiler. Consequently, this type is treated as a basic type like `Int` or `Bool` on which however no basic operations are pre-defined. Type checking of function applications involving these types is performed as usual.

External types are a way to manipulate objects whose structure does not correspond to the type scheme of the D language. For example, the SACLIB user guide says [BCE+92]:

> The SACLIB representation `I` of a number $n \in \mathbf{Z}$ is defined as follows:
>
> - If $\texttt{-BETA} < n < \texttt{BETA}$ then `I` is the integer whose value is n.
> - If $n \leq \texttt{-BETA}$ or $\texttt{BETA} \leq n$ then `I` is the list $(d_0, d_1, \ldots, d_k)$ with $d_k \neq 0, d_i \leq 0$ if $n < 0$ and $0 \leq d_i$ if $0 < n$ for $0 \leq i \leq k$ and $n = \sum_{i=0}^{k} d_i \texttt{BETA}^i$.

This representation does not match any valid D type, consequently we define

```
extern type Integer
```

and do (within a D program) not care for the type's representation.

6.4 External Values

The definition of a D program as a closed expression is very restrictive since it allows to implement only *programs* but not *library functions*. A solution to this problem is to define variables as external:

```
extern x :: Integer
extern y :: Bool
```

In this case, the compiler treats these variables as normal constants (whose definitions are however missing) but finally generates a main function that contains the names of these constants as formal parameters:

```
Integer dcmain(Integer x, Bool y) {
  ...
}
```

The order of the parameters reflects the order of the external declarations. This is the only place where the order of definitions in a D program does matter; to avoid errors, external variable declarations should therefore occur only at a single place at the top-level of the program (but this is not checked by the compiler).

There is a second form of external variable definitions:

```
extern const PRIME :: [Integer]
```

A `const` definition is semantically equivalent to an external variable definition, but the compiler generates a different form of target code:

```
extern WORD PRIME;
Integer dcmain(Integer x, Bool y) {
  ...
}
```

Instead of a parameter of the main function, `PRIME` is a global variable that is given its value in another object file. This form of declarations is mainly useful for variables that are defined once and not changed any more.

6.5 External Functions

A D program may also call human-written C functions respectively C functions that have been automatically generated from other D programs. E.g. the declaration

```
extern IPROD :: (Integer, Integer) -> Integer
```

allows the SACLIB function `IPROD` to be called in a D program. This gives the possibility to reuse existing C source code and also to implement functions whose performance is critical directly in C.

The use of external functions is potentially hazardous in two respects:

1. *Function Interface:* It is entirely the responsibility of the programmer to ensure that the actual function interface matches the declaration in the D program and that the function obeys the calling conventions of D programs. In particular, if the result type of an external function is not an external type, the data representation of the result must match the representation used by the D program.

2. *Referential Transparency:* It is entirely the responsibility of the programmer to ensure that the external function is referentially transparent with respect to the functional layer. The compiler may assume that an external function always returns the same result when given the same input[3].

[3] In Appendix B.4 we give the source code of a D program that uses the SACLIB random generator `IRAND`. However, since l is constant, the compiler is free to replace each call of `IRAND(`l`)` by the same constant such that all "random" numbers become equal.

> Moreover, it is even dangerous, if an external function causes side-effects that are not visible from the functional program (like output): the order in which these side-effects occur is not defined beyond the restrictions imposed by the denotational semantics given in Section 2.

There is a special external declaration for functions that return multiple arguments:

```
extern proc IQR :: (Integer, Integer) -> (Integer, Integer)
```

The `proc` declaration says that `IQR` is not a function that returns a tuple result but a procedure that expects additionally to the input arguments the addresses of two variables to store the results in:

```
void IQR(Word a, Word b, Word *q, Word *r)
```

This form of declarations is only allowed for tuple results (of at least two components); the compiler optionally generates this form of interface also for the main function of a D program.

7 Benchmarks

In this section, we present some benchmark results that demonstrate the efficiency of the generated code.

First, we give some of the "toy" benchmarks that seem to be standard for comparing implementations of functional languages:

- `primes`(n) computes the first n prime numbers using a simple recursive algorithm without caring for any particular optimizations. Nevertheless the D compiler transforms all recursive functions into loops.
- `queens`(n) solves the n-queens problems using two mutually recursive functions. The D compiler transforms the recursion cycle into a single function with two directly recursive calls and transforms one of these recursive calls into loop iteration.
- `qsort`(n) sorts a list of n random numbers using the Quicksort algorithm in a version that copes without an explicit append operation. The D compiler transforms one of the recursive calls into loop iteration.

The timings were performed on a DECstation 5000/25Mhz under Ultrix 4.0 where the generated C code was compiled with the GNU C compiler. The results are compared with the New Jersey implementation of Standard ML Release 0.93 which is a modern implementations of a strict functional language [AM91]. Garbage collection times are excluded.

Benchmark	n	D	SML/NJ 0.93
primes	15000	2580 ms	2550 ms
queens	11	1690 ms	2750 ms
qsort	2500	5350 ms	9150 ms

Basically the D implementation is more than 1.5 times faster than SML/NJ. However, in the `primes` example with comparatively few list constructions but many list accesses, the indirect list referencing scheme of the SACLIB runtime system appears to degrade the performance.

As an example of a more realistic application, we present in Appendix B the full source code of a D program for the exact solution of a linear equation system with integer coefficients. The program is based on a modular algorithm and uses a C subroutine for the computation of determinants by L/U decomposition.

The algorithms applied in the D/C functions are highly optimized. The timings are compared with a program that was entirely written in SACLIB C using the same algorithm (actually the D program was derived from the original SACLIB solution). n is the dimension of the system (i.e. the number of variables) and l the bit-length of its (random) coefficents.

The timings were performed on a Sequent Symmetry 80386/20Mhz under Dynix/ptx 2.0.

Benchmark	n/l	D	SACLIB 1.1
linsolve	5/512	4810 ms	4830 ms
linsolve	10/256	10500 ms	10510 ms
linsolve	40/016	20910 ms	20950 ms

It is not clear to us why the D program is actually slightly *faster* than the original SACLIB solution. One reason may be that the manually coded C program is totally based on SACLIB arrays which are less efficiently implemented than SACLIB lists. Perhaps also the fact that the automatically generated code is very compact makes it more amenable to the optimizations performed by the C compiler.

Appendix C gives two variations that reflect the transformations that were performed for a parallel PACLIB implementation of the algorithm. The timings for these programs are identical to the figures given above.

8 Relationship to Other Work

In this section, we relate our work to the results of other groups.

There are a few compilers for Lisp and functional languages that generate C code:

The Alfa language [GH87, Gol88] is compiled into C code for hypercube systems but the absolute performance of programs is rather low. [Bar89] introduces SCHEME–>C, a portable Scheme compiler; the generated code performs well compared to VAX Common Lisp. [TAL90] describes a compiler for Standard ML that uses C as a compilation target. The Glasgow Haskell compiler [PJHH+93] is based on the Spineless Tagless G-Machine [Pey91] that uses C as a sort of portable macro-assembler. The compiler for the strict functional language Opal [Pep92] generates very efficent C target code.

In most of these approaches the main motivation for choosing C as a compilation target is to guarantee the portability of the compiler to different target architectures. In general the generated code can not be easily embedded into application programs.

The transformation of tail-recursive functions into loops is a standard technique performed in all optimizing Lisp and Scheme compilers e.g. [Kra88]. There is also a considerable amount of literature on all aspects of the compilation of (in particular lazy) functional languages, e.g. [PJ87], [Joh87], [Aug87], [FH88] and [PL92].

The iterated compositions of D are essentially a combination of the higher-order operator `fold` with (a generalized form of) the notion of list comprehensions.

List comprehensions were introduced in KHC and its successor MIRANDA [Tur85] to construct lists without explicitly applying recursion. [PJ87] describes how to transform list comprehensions into recursive functions. The concept was later extended to array comprehensions in Id [Nik88] and taken over by HASKELL [HPW+92]. Id also provides loops as a syntactic sugar for expressing tail-recursive functions.

While the basic idea of iterated compositions is to express computational patterns without constructing an intermediate data structure, one can simulate their effect by applying `fold` to the result list of a comprehension. Recently there has been substantial research how to eliminate such forms of intermediate lists.

[Wad90] has devised a transformation to eliminate intermediate lists that are only used as "glue" to connect separate parts of a program together. However, the applicability of the algorithm is limited to restricted forms of function definitions.

Recently, [GLPJ93] has proposed a more practical approach to the problem that is able to eliminate many of these intermediate lists. The technique extends straightforwardly to improve the compilation of list comprehensions and has been implemented in the Glasgow Haskell compiler.

[Wat91] has introduced a technique that transforms in imperative programs series expressions on linear data structures into loops and fuses multiple walks down the same data structure.

The application of the `fold` operator for expressing computational patterns has been suggested by [MFP91]. [SF93] describes a normalization algorithm that improves programs that are expressed by fold operations by eliminating multiple traverses of data structures and the creation of intermediate lists.

9 Conclusions and Further Work

By the design and implementation of the D language, we have a useable platform for our further work on a parallel functional programming system for computer algebra. The language is simple but rich enough for the implementation of algebraic algorithms and the compiler generates reasonably optimized C code.

A key feature that is missing and will be eventually added is the data type "array" and the extension of iterated compositions to array objects. Overloading and a module concept are also probable candidates for further extensions.

However the next steps will be the extension of the language by support for parallel programming in a *para-functional* style [Hud86]:

1. A unary type constructor `Task` and the operations

   ```
   start   :: (*a->*b, *a) -> Task(*b)
   wait    :: [Task(*a)] -> (*a, Task(*a), [Task(*a)])
   waitone :: Task(*a) -> *a
   stop    :: [Task(*a)] -> Bool
   ```

   that will be directly translated into the corresponding PACLIB functions `pacStart`, `pacWait`, `pacWaitOne`, and `pacStop`.

2. A unary type constructor `Stream` and constructs that correspond to the list constructors/selectors `[ ]`, `:` and `case` and will be compiled into the PACLIB operations `pacOpen`, `pacGet`, `pacPut` and `pacClose`.

3. A parallelization annotation

   exp`@`*cond*

   that starts a parallel task for the evaluation of *exp* provided that *cond* yields true. This annotation will be the basis of para-functional programming and be translated by the compiler into appropriate `Task` definitions and applications of `start` and `wait`.

4. A corresponding annotation for list-valued tasks whose results shall be constructed in a stream-wise fashion.

5. The extension of the notion of iterated compositions to *parallel compositions* that are automatically translated into `@` annotations.

After the implementation of this *para-functional* variant of D, we will pursue the exploitation of algebraic program properties for automatic program restructuring and the automatic time analysis of D programs.

References

[AM91] Andrew Appel and David MacQueen. Standard ML of New Jersey. Release notes (version 0.75), AT&T Bell Laboratories, Murray Hill, NJ 07974, USA, November 1991.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Bar89] Joel F. Bartlett. Scheme->C — A Portable Scheme-to-C Compiler. Technical report, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, California, 1989.

[BCE+92] Bruno Buchberger, George Collins, Mark Encarnation, Hoon Hong, Jeremy Johnson, Werner Krandick, Rüdiger Loos, Ana Mandache, Andreas Neubacher, and Herbert Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

[GH87] Benjamin Goldberg and Paul Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In *Graph Reduction and Techniques, Proceedings of the Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 94–113. Springer, Berlin, 1987.

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

[Gol88] Benjamin F. Goldberg. Multiprocessor Execution of Functional Programs. Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[HPW+92] Paul [editor] Hudak, Simon [editor] Peyton Jones, Philip [editor] Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Research Report 1.2, Department of Computer Science and Department of Computing Science, Yale University, New Haven, Conneticut, U.S.A. and University of Glasgow, Glasgow, U.K., March 1992.

[HSN+92] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report ACPC/TR 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[Hud86] Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages, Part I.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Kra88] David Andrew Kranz. ORBIT: An Optimizing Compiler for Scheme. Technical report, Yale University, New Haven, Connecticut, 1988.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer, Berlin.

[Nik88] Rishiyur S. Nikhil. ID (Version 88.0) Reference Manual. Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

[Pep92] Peter Pepper. The Programming Language Opal. Technical report, Opal Group, Fachbereich Informatik, Technical University Berlin, December 1992.

[Pey91] Simon L. Peyton Jones. The Spineless Tagless G-machine: second attempt. Technical report, Department of Computing Science, University of Glasgow, June 1991.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, New York, 1987.

[PJHH+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, Keele, 1993.

[PL92] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a tutorial.* Prentice Hall, 1992.

[Sch86] David A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[SF93] Tim Sherard and Leonidas Fegaras. A Fold for All Seasons. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

[Sta92] Volker Stahl. Solving a System of Linear Equations with Modular Arithmetic on a MIMD Computer. Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-5, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No Assembly Required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[Tur85] D. A. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Wad90] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wat91] R. Waters. Automatic Transformation of Series Expressions into Loops. *Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

A Concrete Syntax of D

A.1 Lexical Syntax

```
/*****************************************************************************
 *
 * D lexical syntax (UNIX lex style, actually GNU flex)
 *
 *****************************************************************************/

letter [a-zA-Z]
digit  [0-9]
letdig [a-zA-Z0-9]
strch  [^\"]

ident  {letter}{letdig}*
number {digit}{digit}*
string \"{strch}*\"

"let"    return(LETOP);
"where"  return(WHEREOP);
"in"     return(INOP);
"if"     return(IFOP);
"then"   return(THENOP);
"else"   return(ELSEOP);
"case"   return(CASEOP);
"of"     return(OFOP);
"type"   return(TYPEOP);
"loop"   return(LOOPOP);
"list"   return(LISTOP);
"comp"   return(COMPOP);
"rlist"  return(RLISTOP);
"rcomp"  return(RCOMPOP);
"extern" return(EXTERNOP);
"proc"   return(PROCOP);
"const"  return(CONSTOP);

"="    return(DEFOP);
"->"   return(FUNARROP);
"=>"   return(CASEARROP);
":"    return(CONSOP);
"::"   return(SIGOP);
";"    return(SEMICOLOP);
","    return(COMMAOP);
"."    return(DOTOP);
"|"    return(BAROP);
"<-"   return(GENARROP);
".."   return(DOTSOP);
"+="   return(BYOP);
"-="   return(DOWNBYOP);
"'"    return(PRIMEOP);

"("    return(LPAROP);
")"    return(RPAROP);
"{"    return(LBRACEOP);
"}"    return(RBRACEOP);
"["    return(LBRACKOP);
"]"    return(RBRACKOP);

"+"    return(ADDOP);
"-"    return(SUBOP);
"*"    return(MULOP);
"/"    return(DIVOP);
"%"    return(MODOP);
"&&"   return(ANDOP);
"||"   return(OROP);
"~"    return(NOTOP);
```

```
"=="   return(EQOP);
"/="   return(NEQOP);
"<"    return(LTOP);
"<="   return(LEQOP);
">"    return(GTOP);
">="   return(GEQOP);

"True"  return(TRUEOP);
"False" return(FALSEOP);
"Nil"   return(NILOP);
"Int"   return(INTOP);
"Bool"  return(BOOLOP);
"Word"  return(WORDOP);
"Error" return(ERROROP);

{number} return(NUMBER);
{ident}  return(IDENT);
{string} return(STRING);
```

A.2 Parser Syntax

```
/*****************************************************************************
 *
 * D parser syntax (UNIX yacc style, actually GNU bison)
 *
 *****************************************************************************/

/*****************************************************************************
 * terminal symbols
 *****************************************************************************/

%token LETOP IFOP THENOP ELSEOP CASEOP OFOP TYPEOP INOP
%token LOOPOP LISTOP COMPOP RLISTOP RCOMPOP EXTERNOP WHEREOP PROCOP CONSTOP
%token DEFOP FUNARROP CASEARROP CONSOP SIGOP SEMICOLOP COMMAOP DOTOP
%token BAROP GENARROP DOTSOP BYOP DOWNBYOP ATOP PRIMEOP
%token LPAROP RPAROP LBRACEOP RBRACEOP LBRACKOP RBRACKOP
%token ADDOP SUBOP MULOP DIVOP MODOP ANDOP OROP NOTOP
%token EQOP NEQOP LTOP LEQOP GTOP GEQOP
%token TRUEOP FALSEOP NILOP INTOP BOOLOP WORDOP ERROROP
%token NUMBER IDENT STRING
%token EOFOP

/****************************************************************************
 * precedences and associativities
 ****************************************************************************/

%nonassoc LETOP WHEREOP
%right    OROP
%right    ANDOP
%nonassoc EQOP NEQOP LTOP LEQOP GTOP GEQOP
%right    CONSOP
%left     ADDOP SUBOP
%left     MULOP DIVOP MODOP
%nonassoc NOTOP
%left     DOTOP
%left     ATOP
%left     FUNARROP

/*****************************************************************************
 * nonterminal symbols
 *****************************************************************************/

%type <enode> program expression blockexp condexp lcaseexp basicexp
%type <enode> compexp parexp listexp listexp0
%type <elist> explist
%type <snode> statement
%type <slist> statseq
```

```
%type <tnode> type
%type <tlist> typelist
%type <cnode> composer
%type <nnode> nextstat
%type <nlist> nextseq
%type <lnode> loop
%type <llist> looplist
%type <gnode> generator
%type <glist> genlist
%type <ilist> identlist tidentlist
%type <ilist> pattern
%type <plist> patternlist
%type <tokval> niltag semicols

/******************************************************************************
 * parser rules
 ******************************************************************************/

program:
  expression EOFOP
;

statseq:
  statement semicols statseq
|
;

semicols:
  semicols SEMICOLOP
|
;

statement:
  IDENT SIGOP type
| EXTERNOP IDENT SIGOP type
| EXTERNOP PROCOP IDENT SIGOP type
| EXTERNOP CONSTOP IDENT SIGOP type
| EXTERNOP TYPEOP IDENT
| TYPEOP IDENT DEFOP type
| TYPEOP IDENT LPAROP tidentlist RPAROP DEFOP type
| IDENT DEFOP expression
| IDENT LPAROP patternlist RPAROP DEFOP expression
| pattern DEFOP expression
| error
| EOFOP
;

pattern:
  LPAROP identlist RPAROP { $$ = $2 }
;

patternlist:
  pattern COMMAOP patternlist
| IDENT COMMAOP patternlist
| pattern
| IDENT
;

tidentlist:
  MULOP IDENT COMMAOP tidentlist
| MULOP IDENT
;

identlist:
   IDENT COMMAOP identlist
|  IDENT
;

type:
```

```
  INTOP
| BOOLOP
| WORDOP
| LBRACKOP type RBRACKOP
| LPAROP typelist RPAROP
| LPAROP RPAROP FUNARROP type
| type FUNARROP type
| IDENT
| IDENT LPAROP typelist RPAROP
| MULOP IDENT
;

typelist:
  type COMMAOP typelist
| type
;

expression:
  blockexp
| condexp
| lcaseexp
| compexp
| parexp
| basicexp
| listexp
;

blockexp:
  LETOP LBRACEOP statseq RBRACEOP INOP expression
| expression WHEREOP LBRACEOP statseq RBRACEOP
;

condexp:
  IFOP expression THENOP expression ELSEOP expression
;

lcaseexp:
  CASEOP expression OFOP LBRACEOP
  niltag CASEARROP expression semicols
  IDENT CONSOP IDENT CASEARROP expression semicols
  RBRACEOP
;

niltag:
  NILOP
| LBRACKOP RBRACKOP
;

compexp:
  composer LBRACKOP expression BAROP looplist RBRACKOP
;

composer:
  LOOPOP LPAROP expression RPAROP
| LISTOP
| RLISTOP
| COMPOP LPAROP expression COMMAOP expression RPAROP
| RCOMPOP LPAROP expression COMMAOP expression RPAROP
;

/* loop list */
looplist:
  loop BAROP   looplist
| loop
|
;

loop:
  genlist
```

```
| SEMICOLOP expression
| SEMICOLOP SEMICOLOP nextseq
| genlist SEMICOLOP expression
| genlist SEMICOLOP SEMICOLOP nextseq
| SEMICOLOP expression SEMICOLOP nextseq
| genlist SEMICOLOP expression SEMICOLOP nextseq
;

nextstat:
  statement
| IDENT PRIMEOP DEFOP expression
;

nextseq:
  nextstat COMMAOP nextseq
| nextstat
|
;

genlist:
  generator COMMAOP genlist
| generator
|
;

generator:
  IDENT GENARROP expression DOTSOP expression BYOP expression
| IDENT GENARROP expression DOTSOP expression DOWNBYOP expression
| IDENT GENARROP expression DOTSOP expression
| IDENT GENARROP expression BYOP expression
| IDENT GENARROP expression DOWNBYOP expression
| IDENT GENARROP expression DOTSOP
| IDENT GENARROP expression
| IDENT CONSOP IDENT GENARROP expression
| statement
;

basicexp:
  NOTOP expression
| SUBOP expression
| expression EQOP  expression
| expression NEQOP expression
| expression LTOP  expression
| expression LEQOP expression
| expression GTOP  expression
| expression GEQOP expression
| expression ADDOP expression
| expression SUBOP expression
| expression MULOP expression
| expression DIVOP expression
| expression MODOP expression
| expression ANDOP expression
| expression OROP  expression
| expression CONSOP expression
| NILOP
| TRUEOP
| FALSEOP
| NUMBER
| IDENT
| LPAROP explist RPAROP
| IDENT LPAROP explist RPAROP
| IDENT LPAROP RPAROP
| ERROROP LPAROP STRING RPAROP
;

listexp:
  LBRACKOP RBRACKOP
| LBRACKOP listexp0 RBRACKOP
;
```

```
listexp0:
  expression COMMAOP listexp0
| expression
;

explist:
  expression COMMAOP explist
| expression
;
```

B Application Example

As an example for a "real-world" application implemented in the D language we give a full program that also shows the interaction between D and SACLIB C. This program computes the exact solution of a system of linear equations with arbitrary size integer coefficients.

The basic idea is to apply Cramer's rule to transform the original problem into a problem of determinant computation. Determinants are computed by a modular algorithm that maps the problem from the infinite integer domain into several finite modular domains and computes the modular determinants. The Chinese remainder algorithm is applied to lift the modular determinants back into integer determinants from which the solutions of the equation system are computed. The algorithm is described in detail in [Sta92].

We implement a very efficient version of the algorithm that is carefully tuned in order not to compute any required values more than once (as it is done in simple versions of the algorithm). The program consists of several files:

1. The `Makefile` for the program.
2. `linSolve.d` contains the core algorithm.
3. `linCheck.d` checks the correctness of the solution.
4. `linRand.d` generates a random equation system
5. `linDets.c` computes the modular determinants.
6. `linMain.c` contains the input/output interface.

Before compilation, a D source file is automatically passed through the GNU C preprocessor that resolves all `#include` references by including library functions and external signatures.

For each D program the C code generated by the compiler is listed. The algorithm for determinant computation iteratively updates of a two-dimensional array and thus has been implemented directly in C. The generation of a random system has been implemented in D applying the SACLIB random generator, which is a non-functional but nevertheless comfortable "hack".

B.1 `Makefile`

```
###############################################################################
#
# Makefile for linMain
#
# Compiler usage: d2c [options] file
# Options: -e (emit external prototypes)
#          -h (help, print this message)
#          -i (inline list constructor functions)
#          -m mainid (name external main function)
#          -p (print expression before code generation)
```

```
#         -q (print expression after each transformation)
#         -r (print addresses of syntax tree nodes)
#         -s (print symbol table information)
#         -t (pointer arguments from program result tuple)
#         -u (use standalone UNIX runtime system)
#         -v (verbose, print compilation statistics)
#
###############################################################################

# saclib root directory
SACLIB = /usr/local/saclib/saclib1.1

# dc library directory, must also set environment variable DCLIBS to "-I ..."
DCLIB = /usr/local/paclib/dc/lib

# compilation
CC = gcc
CFLAGS = -O2 -I ${SACLIB}/include -I ${DCLIB}

all: linmain

linsolve.c: linsolve.d
	d2c -e -m linSolve linsolve.d

lincheck.c: lincheck.d
	d2c -e -m linCheck lincheck.d

linrand.c: linrand.d
	d2c -t -e -m linRand linrand.d

linmain: linsolve.o linrand.o lincheck.o lindets.o main.o
	${CC} linsolve.o linrand.o lincheck.o lindets.o main.o \
	${SACLIB}/lib/saclibo.a -o linmain
```

B.2 linSolve.d

```
-------------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("depth-first" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
-------------------------------------------------------------------------------
linSolve(A, b, n) where {

  -- program arguments
  extern A :: IntMatrix
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
```

```
  linSolve :: (IntMatrix, IntVector, Digit) -> RatVector
  linSolve(A, b, n) = x where

    B = bound(A, b, n)
    base = (DTOI(1), list [ DTOI(1) | i <- 1..n ])

    (d, ys) = rcomp(cra, base) [ (d, y, p, P) |
        p <- PRIME, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = if d /= 0 then IPROD(DTOI(p), P) else P,
        (d, y) = linDets(A, b, n, p)
     |; d /= 0 ]

    x = list [ RNRED(y, d) | y <- ys ]

  -- prime bound: P = 1+n!*m^n where m = max(Aij, bi)
  bound :: (IntMatrix, IntVector, Digit) -> Integer
  bound(A, b, n) = P where
    m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
    m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
    P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

  -- modular computation of determinants
  extern proc
  linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)

  -- chinese remainder algorithm
  type IIV = (Integer, IntVector)
  cra :: ((Digit, DigitList, Digit, Integer), IIV) -> IIV
  cra((d, ys, p, P), (D0, Y0)) = (D1, Y1) where
    i = MDINV(p, MDHOM(p, P))
    D1 = MIDCRA(P, p, i, D0, d)
    Y1 = list[ MIDCRA(P, p, i, Y, y) | y <- ys, Y <- Y0 ]

}
-------------------------------------------------------------------------------
-- end of linSolve.d
-------------------------------------------------------------------------------

/*
 * Generated by DC ($Revision: 1.6 $ $Date: 1993/08/20 21:19:55 $)
 */

/* #define DC_INLINE */
/* #define DC_UNIX */

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_347;
typedef WORD TUPLE_348;
... omitted ...
typedef WORD TUPLE_554;

/*
 * external functions and constants
 */

Integer IDP2(Integer, INT);
Integer MIPROD(Integer, Integer, Integer);
... omitted ...
Integer IORD2(INT);

/*
 * local function prototypes
 */
```

```
static inline Integer loop_616(IntegerList_97, Integer);
static inline Integer loop_627(IntegerList_97, Integer);
static inline Integer loop_622(IntMatrix_100, Integer);
static inline IntegerList_97 loop_629(INT, INT);
static inline IntegerList_97 loop_610(LIST_347, IntegerList_97, INT, Integer,
  INT);
static inline IIV_104 loop_635(LIST_347, Integer, Integer, IntegerList_97,
  IntMatrix_100, IntegerList_97, INT, Integer);
static inline RatVector_101 loop_643(IntegerList_97, Integer);

/*
 * local function definitions
 */

static inline Integer loop_616(IntegerList_97 xl_613, Integer _615)
{
  while (!ISNIL(xl_613)) {
    _615 = IMAX(IABSF(HEAD(xl_613)), _615);
    xl_613 = TAIL(xl_613);
  }
  return (_615);
}

static inline Integer loop_627(IntegerList_97 xl_624, Integer _626)
{
  while (!ISNIL(xl_624)) {
    _626 = IMAX(IABSF(HEAD(xl_624)), _626);
    xl_624 = TAIL(xl_624);
  }
  return (_626);
}

static inline Integer loop_622(IntMatrix_100 ll_619, Integer _621)
{
  while (!ISNIL(ll_619)) {
    _621 = loop_627(HEAD(ll_619), _621);
    ll_619 = TAIL(ll_619);
  }
  return (_621);
}

static inline void loop_842(INT i_598, INT n_647, IntegerList_97 * _843)
{
  while (i_598 <= n_647) {
    _843 = RCONS(DTOI(1), _843);
    i_598 = (i_598 + 1);
  }
  *_843 = NIL;
  return;
}

static inline IntegerList_97 loop_629(INT i_598, INT n_647)
{
  IntegerList_97 _843;
  loop_842(i_598, n_647, &_843);
  return (_843);
}

static inline void loop_844(LIST_347 yl_605, IntegerList_97 Yl_606, INT p_658,
  Integer P_659, INT i_660, IntegerList_97 * _845)
{
  while (!ISNIL(yl_605))
    if (ISNIL(Yl_606)) {
      *_845 = NIL;
      return;
    } else {
      _845 = RCONS(MIDCRA(P_659, p_658, i_660, HEAD(Yl_606), HEAD(yl_605)),
        _845);
```

```
      Yl_606 = TAIL(Yl_606);
      yl_605 = TAIL(yl_605);
    }
  *_845 = NIL;
  return;
}

static inline IntegerList_97 loop_610(LIST_347 yl_605, IntegerList_97 Yl_606,
  INT p_658, Integer P_659, INT i_660)
{
  IntegerList_97 _845;
  loop_844(yl_605, Yl_606, p_658, P_659, i_660, &_845);
  return (_845);
}

static inline void loop_846(LIST_347 pl_632, Integer P_600, Integer _637,
  IntegerList_97 _638, IntMatrix_100 A_649, IntegerList_97 b_650, INT n_651,
  Integer B_652, Integer * _847, IntegerList_97 * _848)
{
  while (!ISNIL(pl_632))
    if (ICOMP(P_600, B_652) == -1) {
      INT _849;
      LIST_347 _850;
      INT p_827 = HEAD(pl_632);
      linDets(A_649, b_650, n_651, p_827, &_849, &_850);
      if (_849 != 0) {
        INT i_830 = MDINV(p_827, MDHOM(p_827, P_600));
        _638 = loop_610(_850, _638, p_827, P_600, i_830);
        _637 = MIDCRA(P_600, p_827, i_830, _637, _849);
        P_600 = IPROD(DTOI(p_827), P_600);
        pl_632 = TAIL(pl_632);
      } else
        pl_632 = TAIL(pl_632);
    } else {
      *_847 = _637;
      *_848 = _638;
      return;
    }
  *_847 = _637;
  *_848 = _638;
  return;
}

static inline IIV_104 loop_635(LIST_347 pl_632, Integer P_600, Integer _637,
  IntegerList_97 _638, IntMatrix_100 A_649, IntegerList_97 b_650, INT n_651,
  Integer B_652)
{
  IntegerList_97 _847;
  Integer _848;
  loop_846(pl_632, P_600, _637, _638, A_649, b_650, n_651, B_652, &_847, &_848);
  return (TUPLE2(_847, _848));
}

static inline void loop_851(IntegerList_97 yl_640, Integer d_645,
  RatVector_101 * _852)
{
  while (!ISNIL(yl_640)) {
    _852 = RCONS(RNRED(HEAD(yl_640), d_645), _852);
    yl_640 = TAIL(yl_640);
  }
  *_852 = NIL;
  return;
}

static inline RatVector_101 loop_643(IntegerList_97 yl_640, Integer d_645)
{
  RatVector_101 _852;
  loop_851(yl_640, d_645, &_852);
  return (_852);
```

```
}

/*
 * global entry function
 */

RatVector_101 linSolve(IntMatrix_100 A, IntegerList_97 b, INT n)
{
  Integer _853;
  IntegerList_97 _854;
  loop_846(PRIME, DTOI(1), DTOI(1), loop_629(1, n), A, b, n, IPROD(DTOI(2),
    IPROD(IFACTL(n), IEXP(loop_622(A, loop_616(b, DTOI(0))), n))), &_853,
    &_854);
  return (loop_643(_854, _853));
}

/*
 * End of file
 */
```

B.3 linCheck.d

```
--------------------------------------------------------------------------------
--
-- r = linCheck(A, x, b)
--
-- Check the solution of a linear equation system
--
-- A: an Integer Matrix
-- x: a Rational Vector
-- b: an Integer Vector
--
-- r: true iff Ax = b
--
--------------------------------------------------------------------------------
linCheck(A, x, b) where {

  extern A :: IntMatrix
  extern x :: RatVector
  extern b :: IntVector

  #include "dclib.i"
  #include "saclib.i"

  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- returns true iff Ax = b
  linCheck :: (IntMatrix, RatVector, IntVector) -> Bool
  linCheck(A, x, b) = comp(cand, True) [ okay |
    Ai <- A, bi <- b,
    okay = RNCOMP(RNINT(bi), yi) == 0,
    yi = irvprod(Ai, x) ]

  -- integer rational vector product
  irvprod :: (IntVector, RatVector) -> Rational
  irvprod(a, b) =
    rcomp(RNSUM, RNZERO) [ RNPROD(RNINT(ai), bi) | ai <- a, bi <- b ]

  -- would be nice to have this constant in SACLIB
  RNZERO = RNINT(DTOI(0))

}
--------------------------------------------------------------------------------
-- end of linCheck.d
--------------------------------------------------------------------------------
```

```
/*
 * Generated by DC ($Revision: 1.6 $ $Date: 1993/08/20 21:19:55 $)
 */

/* #define DC_INLINE */
/* #define DC_UNIX */

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_311;
typedef WORD TUPLE_312;
... omitted ...
typedef WORD TUPLE_513;

/*
 * external functions and constants
 */

Rational RNQ(Rational, Rational);
Integer IEXP(Integer, INT);
... omitted ...
void IPOWER(Integer, Integer, Integer *, INT *);

/*
 * local function prototypes
 */

static inline Rational loop_548(IntegerList_91, RatVector_95, Rational);
static inline BOOL loop_554(IntMatrix_94, IntegerList_91, RatVector_95,
  Rational);

/*
 * local function definitions
 */

static inline Rational loop_548(IntegerList_91 ail_543, RatVector_95 bil_544,
  Rational _547)
{
  while (!ISNIL(ail_543))
    if (ISNIL(bil_544))
      return (_547);
    else {
      _547 = RNSUM(RNPROD(RNINT(HEAD(ail_543)), HEAD(bil_544)), _547);
      bil_544 = TAIL(bil_544);
      ail_543 = TAIL(ail_543);
    }
  return (_547);
}

static inline BOOL loop_554(IntMatrix_94 Ail_549, IntegerList_91 bil_550,
  RatVector_95 x_556, Rational RNZERO_557)
{
  while (!ISNIL(Ail_549))
    if (ISNIL(bil_550))
      return (TRUE);
    else if (RNCOMP(RNINT(HEAD(bil_550)), loop_548(HEAD(Ail_549), x_556,
      RNZERO_557)) == 0) {
      bil_550 = TAIL(bil_550);
      Ail_549 = TAIL(Ail_549);
    } else
      return (FALSE);
  return (TRUE);
}
```

```
/*
 * global entry function
 */

BOOL linCheck(IntMatrix_94 A, RatVector_95 x, IntegerList_91 b)
{
  return (loop_554(A, b, x, RNINT(DTOI(0))));
}

/*
 * End of file
 */
```

B.4 linRand.d

```
--------------------------------------------------------------------------------
--
-- (A, b) = linRand(n, l, c)
--
-- Generate a random linear equation system
--
-- n: dimension of equation system
-- l: length of random integers in bits
--
-- (A, b): a system of dimension n with integer entries of at most l bits
--
--------------------------------------------------------------------------------
(A, b) where {

  extern n :: Int
  extern l :: Int

  #include "dclib.i"
  #include "saclib.i"

  -- that's a non-functional hack, since the compiler might "optimize" this
  -- code treating IRAND(l) as a constant (fortunately it does not yet :-)
  A = list [ Ai | i <- 1..n, Ai = list [ IRAND(l) | i <- 1..n ] ]
  b = list [ IRAND(l) | i <- 1..n ]

}
--------------------------------------------------------------------------------
-- end of linRand.d
--------------------------------------------------------------------------------
```

```
/*
 * Generated by DC ($Revision: 1.6 $ $Date: 1993/08/20 21:19:55 $)
 */

/* #define DC_INLINE */
/* #define DC_UNIX */

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_300;
typedef WORD TUPLE_301;
... omitted ...
typedef WORD TUPLE_512;

/*
 * external functions and constants
 */
```

```
INT MDCRA(INT, INT, INT, INT, INT);
INT DNOT(INT);
... omitted ...
Integer IBCOEF(INT, INT);

/*
 * local function prototypes
 */

static inline IntegerList_91 loop_520(INT, INT, INT);
static inline IntegerList_91 loop_514(INT, INT, INT);
static inline LIST_507 loop_517(INT, INT, INT);

/*
 * local function definitions
 */

static inline void loop_547(INT i_291, INT n_521, INT l_522,
  IntegerList_91 * _548)
{
  while (i_291 <= n_521) {
    _548 = RCONS(IRAND(l_522), _548);
    i_291 = (i_291 + 1);
  }
  *_548 = NIL;
  return;
}

static inline IntegerList_91 loop_520(INT i_291, INT n_521, INT l_522)
{
  IntegerList_91 _548;
  loop_547(i_291, n_521, l_522, &_548);
  return (_548);
}

static inline void loop_549(INT i_290, INT n_527, INT l_528,
  IntegerList_91 * _550)
{
  while (i_290 <= n_527) {
    _550 = RCONS(IRAND(l_528), _550);
    i_290 = (i_290 + 1);
  }
  *_550 = NIL;
  return;
}

static inline IntegerList_91 loop_514(INT i_290, INT n_527, INT l_528)
{
  IntegerList_91 _550;
  loop_549(i_290, n_527, l_528, &_550);
  return (_550);
}

static inline void loop_551(INT i_288, INT n_524, INT l_525, LIST_507 * _552)
{
  while (i_288 <= n_524) {
    _552 = RCONS(loop_514(1, n_524, l_525), _552);
    i_288 = (i_288 + 1);
  }
  *_552 = NIL;
  return;
}

static inline LIST_507 loop_517(INT i_288, INT n_524, INT l_525)
{
  LIST_507 _552;
  loop_551(i_288, n_524, l_525, &_552);
  return (_552);
}
```

```
/*
 * global entry function
 */

void linRand(INT n, INT l, LIST_507 * _554, IntegerList_91 * _555)
{
  {
    *_554 = loop_517(1, n, l);
    *_555 = loop_520(1, n, l);
    return;
  }
}

/*
 * End of file
 */
```

B.5 linDets.c

```
/******************************************************************************
 *
 * (d, y) = linDets(A, b, n, m)
 *
 * Compute modular determinants
 *
 * A: an Integer matrix of dimension n that is regular
 * b: an Integer vector of length n
 * n: a Digit
 * m: a Digit
 *
 * d: det(A) mod Zm
 * y: list of yi = det(Ai) mod Zm where Ai = A with column i replaced by b
 *
 *****************************************************************************/

#include <saclib.h>

/******************************************************************************
 *
 * Return Am = A mod m and bm = b mod m where Am and bm are arrays!
 *
 *****************************************************************************/
static void linMap(Word A, Word b, Word n, Word m, Word *Am, Word *bm) {

  Word i, j, A_, b_;

  A_ = GCAMALLOC(n, GC_CHECK);
  for (i = 0; i < n; i++)
    GCASET(A_, i, GCAMALLOC(n, GC_CHECK));
  b_ = GCAMALLOC(n, GC_CHECK);

  for (i = 0; i < n; i++) {
    Word Ai = FIRST(A);
    Word A_i = GCAGET(A_, i);
    for (j = 0; j < n; j++) {
      GCASET(A_i, j, MDHOM(m, FIRST(Ai)));
      Ai = RED(Ai);
    }
    GCASET(b_, i, MDHOM(m, FIRST(b)));
    A = RED(A);
    b = RED(b);
  }

  *Am = A_;
  *bm = b_;
```

```
}

/****************************************************************************
 *
 * Compute d = det(A) and yi = det(Ai) in Zm by LU decomposition.
 *
 ****************************************************************************/
void linDets(Word A, Word b, Word n, Word m, Word *d, Word *y) {

  Word Am, bm, ym;
  Word Amk, Amkk, Ami, bmi;
  Word i, j, k, p;
  Word det, tmp;

  linMap(A, b, n, m, &Am, &bm);

  det = 1;
  for (k = 0; k < n-1; k ++) {

    p = k;
    while(p < n && GCAGET(GCAGET(Am, p), k) == 0) p++;
    if (p == n) {
      *d = 0;
      return;
    }

    if (p != k) {
      tmp = GCAGET(Am, k); GCASET(Am, k, GCAGET(Am, p)); GCASET(Am, p, tmp);
      tmp = GCAGET(bm, k); GCASET(bm, k, GCAGET(bm, p)); GCASET(bm, p, tmp);
      det = MDNEG(m, det);
    }

    Amk  = GCAGET(Am, k);
    Amkk = GCAGET(Amk, k);
    det = MDPROD(m, det, Amkk);

    for (i = k+1; i < n; i++) {
      Ami = GCAGET(Am, i);
      GCASET(Ami, k, MDQ(m, GCAGET(Ami, k), Amkk));
      for (j = k+1; j < n; j++) {
        GCASET(Ami, j, MDDIF(m, GCAGET(Ami, j),
                             MDPROD(m, GCAGET(Ami, k), GCAGET(Amk, j))));
      }
    }

  }

  det = MDPROD(m, det, GCAGET(GCAGET(Am, n-1), n-1));
  if (det == 0) {
     *d = 0;
     return;
   }

  for (i = 0; i < n; i++) {
    Ami = GCAGET(Am, i);
    bmi = GCAGET(bm, i);
    for (j = 0; j < i; j++) {
      bmi = MDDIF(m, bmi, MDPROD(m, GCAGET(Ami, j), GCAGET(bm, j)));
    }
    GCASET(bm, i, bmi);
  }

  for (i = n-1; i >= 0; i--) {
    Ami = GCAGET(Am, i);
    tmp = GCAGET(bm, i);
    for (k = i+1; k  < n; k++)
      tmp = MDDIF(m, tmp, MDPROD(m, GCAGET(Ami, k), GCAGET(bm, k)));
    GCASET(bm, i, MDQ(m, tmp, GCAGET(Ami, i)));
  }
```

```
  ym = NIL;
  for (i = n-1; i >= 0; i--)
    ym = COMP(MDPROD(m, det, GCAGET(bm, i)), ym);

  *d = det;
  *y = ym;

}
```

B.6 linMain.c

```
/*****************************************************************************
 *
 * linMain n l c p
 *
 * solve a random linear equation system
 *
 * n: number of dimensions
 * l: length of random integers in bit
 * c: 1 iff result is to be checked
 * p: 1 iff system is to be printed
 *
 *****************************************************************************/

#include <saclib.h>
#include <stdlib.h>

extern void linRand(Word n, Word l, Word *A, Word *b);
extern Word linSolve(Word A, Word b, Word n);
extern Word linCheck(Word A, Word x, Word b);

/*****************************************************************************
 *
 * Get command line arguments
 *
 *****************************************************************************/
void linArgs(int argc, char **argv, Word *n, Word *l, Word *c, Word *p){

  if (argc < 3) {
    SWRITE("syntax: linsolve n l [ c [ p ] ]\n");
    exit(-1);
  }
  *n = atoi(argv[1]);
  *l = atoi(argv[2]);
  if (argc < 4)
    *c = 0;
  else
    *c = atoi(argv[3]);
  if (argc < 5)
    *p = 0;
  else
    *p = atoi(argv[4]);

}

/****************************************************************************
 *
 * print integer list
 *
 ****************************************************************************/
void linPrintI(Word x) {

  SWRITE("\n[");
  while (!ISNIL(x)) {
    IWRITE(FIRST(x));
    x = RED(x);
```

```
    if (!ISNIL(x)) SWRITE(",");
  }
  SWRITE("]");

}

/*****************************************************************************
 *
 * print rational list
 *
 *****************************************************************************/
void linPrintR(Word x) {

  SWRITE("\n[");
  while (!ISNIL(x)) {
    RNWRITE(FIRST(x));
    x = RED(x);
    if (!ISNIL(x)) SWRITE(",");
  }
  SWRITE("]");

}

/******************************************************************************
 *
 * Print result
 *
 ******************************************************************************/
void linPrint(Word A, Word x, Word b) {

  while (!ISNIL(A)) {
    linPrintI(FIRST(A));
    A = RED(A);
  }
  SWRITE("\n");
  linPrintI(b);
  SWRITE("\n");
  linPrintR(x);
  SWRITE("\n");

}

/*******************************************************************************
 *
 * Main program
 *
 *******************************************************************************/
Word sacMain(int argc, char **argv) {

  Word n, l, c, p;
  Word A, x, b;
  Word time;

  linArgs(argc, argv, &n, &l, &c, &p);
  linRand(n, l, &A, &b);

  time = CLOCK();
  x = linSolve(A, b, n);
  time = CLOCK()-time;

  IWRITE(time);
  SWRITE(" ms\n");

  if (p) linPrint(A, x, b);
  if (c && !linCheck(A, x, b)) SWRITE("Something went wrong.\n");

  return(0);

}
```

C Application Example (Variations)

In this Appendix, we give two slight variations of the application example presented before:

- *Depth-First:* In the original version of **linSolve**, the modular determinants were immediately integrated by Chinese remaindering into the preliminary values for the integer determinants.
- *Width-First:* In this version, the modular determinants are stored before further processing. After sufficiently many have been computed, they are combined by Chinese remaindering to yield the integer determinants.
- *Parallel:* In this variant of **linSolve**, the check for unlucky primes is postponed after all determinants have been computed. In the rare case an unlucky prime occurs, the computation is repeated with new primes.

These transformations reflect those that were performed on the original SACLIB implementation of the algorithm to get a parallel PACLIB variant.

```
--------------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("width-first" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
--------------------------------------------------------------------------------
linSolve(A, b, n) where {

  -- program arguments
  extern A :: IntMatrix
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
  linSolve :: (IntMatrix, IntVector, Digit) -> RatVector
  linSolve(A, b, n) = x where

    B = bound(A, b, n)
    nil = []; nil :: DigitList -- satisfy type checker
    base = ([], list [ nil | i <- 1..n ], [], [], [])

    (dk, yk, pk, Pk, ik) = comp(compose, base) [ (d, y, p, P) |
        p <- PRIME, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = if d /= 0 then IPROD(DTOI(p), P) else P,
        (d, y) = linDets(A, b, n, p)
     |; d /= 0 ]

    d = cra(pk, ik, Pk, dk)
    ys = list [ cra(pk, ik, Pk, y) | y <- yk ]
    x = list [ RNRED(y, d) | y <- ys ]
```

```
  -- prime bound: P = 2*n!*m^n where m = max(Aij, bi)
  bound :: (IntMatrix, IntVector, Digit) -> Integer
  bound(A, b, n) = P where
    m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
    m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
    P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

  -- modular computation of determinants
  extern proc
  linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)

  -- composition of modular determinants, primes, products and inverses
  type DIL = (DigitList, [DigitList], DigitList, IntegerList, DigitList)
  compose :: ((Digit, DigitList, Digit, Integer), DIL) -> DIL
  compose((d, y, p, P), (dk, yk, pk, Pk, ik)) =
    (d:dk, Y, p:pk, P:Pk, i:ik) where
      i = MDINV(p, MDHOM(p, P))
      Y = list [ yi:ys | yi <- y, ys <- yk ]

  -- chinese remainder algorithm on list of arguments
  cra :: (DigitList, DigitList, IntegerList, DigitList) -> Integer
  cra(ms, mis, Ms, as) = rcomp(cra2, DTOI(1)) [ (m, mi, M, a) |
      m <- ms, mi <- mis, M <- Ms, a <- as ]

  -- chinese remainder algorithm on two arguments
  cra2 :: ((Digit, Digit, Integer, Digit), Integer) -> Integer
  cra2((m, mi, M, a), A) = MIDCRA(M, m, mi, A, a)

}
------------------------------------------------------------------------------
-- end of linSolve.d
------------------------------------------------------------------------------

/*
 * Generated by DC ($Revision: 1.4 $ $Date: 1993/08/15 20:12:40 $)
 */

/* #define DC_INLINE */
/* #define DC_UNIX */

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_375;
typedef WORD TUPLE_376;

... omitted ...

typedef WORD TUPLE_587;

/*
 * external functions and constants
 */

Integer IDP2(Integer, INT);
void linDets(IntMatrix_102, IntegerList_99, INT, INT, INT *, LIST_375 *);

... omitted ...

Integer IORD2(INT);

/*
 * local function prototypes
 */

static inline Integer loop_693(IntegerList_99, Integer);
```

```
static inline Integer loop_704(IntegerList_99, Integer);
static inline Integer loop_699(IntMatrix_102, Integer);
static inline LIST_378 loop_712(INT, INT);
static inline LIST_378 loop_714(INT, INT);
static inline LIST_378 loop_688(LIST_375, LIST_378);
static inline DIL_106 loop_711(LIST_375, Integer, IntMatrix_102,
  IntegerList_99, INT, Integer);
static inline RatVector_103 loop_748(IntegerList_99, Integer);
static inline Integer loop_739(LIST_375, LIST_375, IntegerList_99, LIST_375,
  Integer);
static inline IntegerList_99 loop_743(LIST_378, LIST_375, IntegerList_99,
  LIST_375);
static inline Integer loop_727(LIST_375, LIST_375, IntegerList_99, LIST_375,
  Integer);

/*
 * local function definitions
 */

static inline Integer loop_693(IntegerList_99 xl_690, Integer _692)
{
  while (!ISNIL(xl_690)) {
    _692 = IMAX(IABSF(HEAD(xl_690)), _692);
    xl_690 = TAIL(xl_690);
  }
  return (_692);
}

static inline Integer loop_704(IntegerList_99 xl_701, Integer _703)
{
  while (!ISNIL(xl_701)) {
    _703 = IMAX(IABSF(HEAD(xl_701)), _703);
    xl_701 = TAIL(xl_701);
  }
  return (_703);
}

static inline Integer loop_699(IntMatrix_102 ll_696, Integer _698)
{
  while (!ISNIL(ll_696)) {
    _698 = loop_704(HEAD(ll_696), _698);
    ll_696 = TAIL(ll_696);
  }
  return (_698);
}

static inline void loop_986(INT i_713, INT n_765, LIST_378 * _987)
{
  while (i_713 <= n_765) {
    _987 = RCONS(NIL, _987);
    i_713 = (i_713 + 1);
  }
  *_987 = NIL;
  return;
}

static inline LIST_378 loop_712(INT i_713, INT n_765)
{
  LIST_378 _987;
  loop_986(i_713, n_765, &_987);
  return (_987);
}

static inline void loop_988(INT i_715, INT n_767, LIST_378 * _989)
{
  while (i_715 <= n_767) {
    _989 = RCONS(NIL, _989);
    i_715 = (i_715 + 1);
  }
```

```
  *_989 = NIL;
  return;
}

static inline LIST_378 loop_714(INT i_715, INT n_767)
{
  LIST_378 _989;
  loop_988(i_715, n_767, &_989);
  return (_989);
}

static inline void loop_990(LIST_375 yil_683, LIST_378 ysl_684, LIST_378
  * _991)
{
  while (!ISNIL(yil_683))
    if (ISNIL(ysl_684)) {
      *_991 = NIL;
      return;
    } else {
      _991 = RCONS(CONS(HEAD(yil_683), HEAD(ysl_684)), _991);
      ysl_684 = TAIL(ysl_684);
      yil_683 = TAIL(yil_683);
    }
  *_991 = NIL;
  return;
}

static inline LIST_378 loop_688(LIST_375 yil_683, LIST_378 ysl_684)
{
  LIST_378 _991;
  loop_990(yil_683, ysl_684, &_991);
  return (_991);
}

static inline void loop_992(LIST_375 pl_708, Integer P_661, IntMatrix_102
  A_756, IntegerList_99 b_757, INT n_758, Integer B_759, LIST_375 * _993,
  LIST_378 * _994, LIST_375 * _995, IntegerList_99 * _996, LIST_375 * _997)
{
  while (!ISNIL(pl_708))
    if (ICOMP(P_661, B_759) == -1) {
      INT _998;
      LIST_375 _999;
      LIST_375 pt_943 = TAIL(pl_708);
      INT p_944 = HEAD(pl_708);
      linDets(A_756, b_757, n_758, p_944, &_998, &_999);
      if (_998 != 0) {
        LIST_378 _1001;
        loop_992(pt_943, IPROD(DTOI(p_944), P_661), A_756, b_757, n_758,
          B_759, RCONS(_998, _993), &_1001, RCONS(p_944, _995), RCONS(P_661,
          _996), RCONS(MDINV(p_944, MDHOM(p_944, P_661)), _997));
        *_994 = loop_688(_999, _1001);
        return;
      } else
        pl_708 = pt_943;
    } else {
      *_993 = NIL;
      *_994 = loop_712(1, n_758);
      *_995 = NIL;
      *_996 = NIL;
      *_997 = NIL;
      return;
    }
  *_993 = NIL;
  *_994 = loop_714(1, n_758);
  *_995 = NIL;
  *_996 = NIL;
  *_997 = NIL;
  return;
}
```

```
static inline DIL_106 loop_711(LIST_375 pl_708, Integer P_661, IntMatrix_102
  A_756, IntegerList_99 b_757, INT n_758, Integer B_759)
{
  LIST_375 _993;
  IntegerList_99 _994;
  LIST_375 _995;
  LIST_378 _996;
  LIST_375 _997;
  loop_992(pl_708, P_661, A_756, b_757, n_758, B_759, &_993, &_994, &_995,
    &_996, &_997);
  return (TUPLE5(_993, _994, _995, _996, _997));
}

static inline void loop_1005(IntegerList_99 yl_745, Integer d_754,
  RatVector_103 * _1006)
{
  while (!ISNIL(yl_745)) {
    _1006 = RCONS(RNRED(HEAD(yl_745), d_754), _1006);
    yl_745 = TAIL(yl_745);
  }
  *_1006 = NIL;
  return;
}

static inline RatVector_103 loop_748(IntegerList_99 yl_745, Integer d_754)
{
  RatVector_103 _1006;
  loop_1005(yl_745, d_754, &_1006);
  return (_1006);
}

static inline Integer loop_739(LIST_375 ml_730, LIST_375 mil_731,
  IntegerList_99 Ml_732, LIST_375 al_733, Integer _738)
{
  while (!ISNIL(ml_730))
    if (ISNIL(mil_731))
      return (_738);
    else if (ISNIL(Ml_732))
      return (_738);
    else if (ISNIL(al_733))
      return (_738);
    else {
      _738 = MIDCRA(HEAD(Ml_732), HEAD(ml_730), HEAD(mil_731), _738,
        HEAD(al_733));
      al_733 = TAIL(al_733);
      Ml_732 = TAIL(Ml_732);
      mil_731 = TAIL(mil_731);
      ml_730 = TAIL(ml_730);
    }
  return (_738);
}

static inline void loop_1008(LIST_378 yl_740, LIST_375 pk_750,
  IntegerList_99 Pk_751, LIST_375 ik_752, IntegerList_99 * _1009)
{
  while (!ISNIL(yl_740)) {
    _1009 = RCONS(loop_739(pk_750, ik_752, Pk_751, HEAD(yl_740),
      DTOI(1)), _1009);
    yl_740 = TAIL(yl_740);
  }
  *_1009 = NIL;
  return;
}

static inline IntegerList_99 loop_743(LIST_378 yl_740, LIST_375 pk_750,
  IntegerList_99 Pk_751, LIST_375 ik_752)
{
  IntegerList_99 _1009;
```

```
  loop_1008(yl_740, pk_750, Pk_751, ik_752, &_1009);
  return (_1009);
}

static inline Integer loop_727(LIST_375 ml_718, LIST_375 mil_719,
  IntegerList_99 Ml_720, LIST_375 al_721, Integer _726)
{
  while (!ISNIL(ml_718))
    if (ISNIL(mil_719))
      return (_726);
    else if (ISNIL(Ml_720))
      return (_726);
    else if (ISNIL(al_721))
      return (_726);
    else {
      _726 = MIDCRA(HEAD(Ml_720), HEAD(ml_718), HEAD(mil_719), _726,
        HEAD(al_721));
      al_721 = TAIL(al_721);
      Ml_720 = TAIL(Ml_720);
      mil_719 = TAIL(mil_719);
      ml_718 = TAIL(ml_718);
    }
  return (_726);
}

/*
 * global entry function
 */

RatVector_103 linSolve(IntMatrix_102 A, IntegerList_99 b, INT n)
{
  LIST_375 _1011;
  LIST_378 _1012;
  LIST_375 _1013;
  IntegerList_99 _1014;
  LIST_375 _1015;
  loop_992(PRIME, DTOI(1), A, b, n, IPROD(DTOI(2), IPROD(IFACTL(n),
    IEXP(loop_699(A, loop_693(b, DTOI(0))), n))), &_1011, &_1012, &_1013,
    &_1014, &_1015);
  return (loop_748(loop_743(_1012, _1013, _1014, _1015), loop_727(_1013,
    _1015, _1014, _1011, DTOI(1))));
}

/*
 * End of file
 */

-------------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("parallel" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
-------------------------------------------------------------------------------
linSolve(A, b, n, PRIME) where {

  -- program arguments
  extern A :: IntMatrix
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
```

```
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
  linSolve :: (IntMatrix, IntVector, Digit, DigitList) -> RatVector
  linSolve(A, b, n, PLIST) = x where

    B = bound(A, b, n)
    nil = []; nil :: [Digit] -- satisfy type checker
    base = ([], list [ nil | i <- 1..n ], [], [], [], PLIST)

    (dk, yk, pk, Pk, ik, PREST) = comp(compose, base) [ (d, y, p, P, PREST) |
        p:PREST <- PLIST, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = IPROD(DTOI(p), P),
        (d, y) = linDets(A, b, n, p) ]

    d = cra(pk, ik, Pk, dk, dk)
    ys = list [ cra(pk, ik, Pk, y, dk) | y <- yk ]
    x0 = list [ RNRED(y, d) | y <- ys ]

    x = if comp(and, True) [ d /= 0 | d <- dk ]
          then x0
          else linSolve(A, b, n, PREST)

  -- prime bound: P = 2*n!*m^n where m = max(Aij, bi)
  bound :: (IntMatrix, IntVector, Digit) -> Integer
  bound(A, b, n) = P where
    m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
    m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
    P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

  -- modular computation of determinants
  extern proc
  linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)

  -- composition of modular determinants, primes, products and inverses
  type DIL = (DigitList,[DigitList],DigitList,IntegerList,DigitList,DigitList)
  compose :: ((Digit, DigitList, Digit, Integer, DigitList), DIL) -> DIL
  compose((d, y, p, P, PREST), (dk, yk, pk, Pk, ik, PREST0)) =
    (d:dk, Y, p:pk, P:Pk, i:ik, PREST) where
      i = MDINV(p, MDHOM(p, P))
      Y = list [ yi:ys | yi <- y, ys <- yk ]

  -- chinese remainder algorithm on list of arguments
  cra :: (DigitList, DigitList, IntegerList, DigitList, DigitList) -> Integer
  cra(ms, mis, Ms, as, ds) = rcomp(cra2, DTOI(1)) [ (m, mi, M, a) |
      m <- ms, mi <- mis, M <- Ms, a <- as, d <- ds
    ; d /= 0 ]

  -- chinese remainder algorithm on two arguments
  cra2 :: ((Digit, Digit, Integer, Digit), Integer) -> Integer
  cra2((m, mi, M, a), A) = MIDCRA(M, m, mi, A, a)

}
-----------------------------------------------------------------------------
-- end of linSolve.d
-----------------------------------------------------------------------------

/*
 * Generated by DC ($Revision: 1.4 $ $Date: 1993/08/15 20:12:40 $)
 */

/* #define DC_INLINE */
/* #define DC_UNIX */
```

```
#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_384;
typedef WORD TUPLE_385;

... omitted ...

typedef WORD TUPLE_597;

/*
 * external functions and constants
 */

Integer IDP2(Integer, INT);
Integer MIPROD(Integer, Integer, Integer);

... omitted ...

/*
 * local function prototypes
 */

static inline Integer loop_651(IntegerList_99, Integer);
static inline Integer loop_662(IntegerList_99, Integer);
static inline Integer loop_657(IntMatrix_102, Integer);
static inline BOOL loop_691(LIST_384);
static inline LIST_387 loop_669(INT, INT);
static inline LIST_387 loop_671(INT, INT);
static inline LIST_387 loop_646(LIST_384, LIST_387);
static inline DIL_106 loop_668(LIST_384, Integer, IntMatrix_102,
  IntegerList_99, INT, LIST_384, Integer);
static inline RatVector_103 loop_714(IntegerList_99, Integer);
static inline Integer loop_705(LIST_384, LIST_384, IntegerList_99, LIST_384,
  LIST_384, Integer);
static inline IntegerList_99 loop_709(LIST_387, LIST_384, LIST_384,
  IntegerList_99, LIST_384);
static inline Integer loop_686(LIST_384, LIST_384, IntegerList_99, LIST_384,
  LIST_384, Integer);
static inline RatVector_103 linSolve_104(IntMatrix_102, IntegerList_99, INT,
  LIST_384);

/*
 * local function definitions
 */

static inline Integer loop_651(IntegerList_99 xl_648, Integer _650)
{
  while (!ISNIL(xl_648)) {
    _650 = IMAX(IABSF(HEAD(xl_648)), _650);
    xl_648 = TAIL(xl_648);
  }
  return (_650);
}

static inline Integer loop_662(IntegerList_99 xl_659, Integer _661)
{
  while (!ISNIL(xl_659)) {
    _661 = IMAX(IABSF(HEAD(xl_659)), _661);
    xl_659 = TAIL(xl_659);
  }
  return (_661);
}

static inline Integer loop_657(IntMatrix_102 ll_654, Integer _656)
```

```
{
  while (!ISNIL(ll_654)) {
    _656 = loop_662(HEAD(ll_654), _656);
    ll_654 = TAIL(ll_654);
  }
  return (_656);
}

static inline BOOL loop_691(LIST_384 dl_688)
{
  if (ISNIL(dl_688))
    return (TRUE);
  else
    return (((HEAD(dl_688) != 0) && loop_691(TAIL(dl_688))));
}

static inline void loop_1001(INT i_670, INT n_732, LIST_387 * _1002)
{
  while (i_670 <= n_732) {
    _1002 = RCONS(NIL, _1002);
    i_670 = (i_670 + 1);
  }
  *_1002 = NIL;
  return;
}

static inline LIST_387 loop_669(INT i_670, INT n_732)
{
  LIST_387 _1002;
  loop_1001(i_670, n_732, &_1002);
  return (_1002);
}

static inline void loop_1003(INT i_672, INT n_734, LIST_387 * _1004)
{
  while (i_672 <= n_734) {
    _1004 = RCONS(NIL, _1004);
    i_672 = (i_672 + 1);
  }
  *_1004 = NIL;
  return;
}

static inline LIST_387 loop_671(INT i_672, INT n_734)
{
  LIST_387 _1004;
  loop_1003(i_672, n_734, &_1004);
  return (_1004);
}

static inline void loop_1005(LIST_384 yil_641, LIST_387 ysl_642, LIST_387
  * _1006)
{
  while (!ISNIL(yil_641))
    if (ISNIL(ysl_642)) {
      *_1006 = NIL;
      return;
    } else {
      _1006 = RCONS(CONS(HEAD(yil_641), HEAD(ysl_642)), _1006);
      ysl_642 = TAIL(ysl_642);
      yil_641 = TAIL(yil_641);
    }
  *_1006 = NIL;
  return;
}

static inline LIST_387 loop_646(LIST_384 yil_641, LIST_387 ysl_642)
{
  LIST_387 _1006;
```

```
  loop_1005(yil_641, ysl_642, &_1006);
  return (_1006);
}

static inline void loop_1007(LIST_384 pl_666, Integer P_324, IntMatrix_102
  A_723, IntegerList_99 b_724, INT n_725, LIST_384 PLIST_726, Integer B_727,
  LIST_384 * _1008, LIST_387 * _1009, LIST_384 * _1010, IntegerList_99
  * _1011, LIST_384 * _1012, LIST_384 * _1013)
{
  if (ISNIL(pl_666)) {
    *_1008 = NIL;
    *_1009 = loop_671(1, n_725);
    *_1010 = NIL;
    *_1011 = NIL;
    *_1012 = NIL;
    *_1013 = PLIST_726;
    return;
  } else if (ICOMP(P_324, B_727) == -1) {
    INT _1014;
    LIST_384 _1015;
    LIST_387 _1017;
    LIST_384 _1021;
    LIST_384 PREST_926 = TAIL(pl_666);
    INT p_927 = HEAD(pl_666);
    linDets(A_723, b_724, n_725, p_927, &_1014, &_1015);
    loop_1007(PREST_926, IPROD(DTOI(p_927), P_324), A_723, b_724, n_725,
      PLIST_726, B_727, RCONS(_1014, _1008), &_1017, RCONS(p_927, _1010),
      RCONS(P_324, _1011), RCONS(MDINV(p_927, MDHOM(p_927, P_324)), _1012),
      &_1021);
    *_1009 = loop_646(_1015, _1017);
    *_1013 = PREST_926;
    return;
  } else {
    *_1008 = NIL;
    *_1009 = loop_669(1, n_725);
    *_1010 = NIL;
    *_1011 = NIL;
    *_1012 = NIL;
    *_1013 = PLIST_726;
    return;
  }
}

static inline DIL_106 loop_668(LIST_384 pl_666, Integer P_324, IntMatrix_102
  A_723, IntegerList_99 b_724, INT n_725, LIST_384 PLIST_726, Integer B_727)
{
  LIST_384 _1008;
  LIST_384 _1009;
  IntegerList_99 _1010;
  LIST_384 _1011;
  LIST_387 _1012;
  LIST_384 _1013;
  loop_1007(pl_666, P_324, A_723, b_724, n_725, PLIST_726, B_727, &_1008,
    &_1009, &_1010, &_1011, &_1012, &_1013);
  return (TUPLE6(_1008, _1009, _1010, _1011, _1012, _1013));
}

static inline void loop_1022(IntegerList_99 yl_711, Integer d_721,
  RatVector_103 * _1023)
{
  while (!ISNIL(yl_711)) {
    _1023 = RCONS(RNRED(HEAD(yl_711), d_721), _1023);
    yl_711 = TAIL(yl_711);
  }
  *_1023 = NIL;
  return;
}

static inline RatVector_103 loop_714(IntegerList_99 yl_711, Integer d_721)
```

```
{
  RatVector_103 _1023;
  loop_1022(yl_711, d_721, &_1023);
  return (_1023);
}

static inline Integer loop_705(LIST_384 ml_694, LIST_384 mil_695,
  IntegerList_99 Ml_696, LIST_384 al_697, LIST_384 dl_698, Integer _704)
{
  while (!ISNIL(ml_694))
    if (ISNIL(mil_695))
      return (_704);
    else if (ISNIL(Ml_696))
      return (_704);
    else if (ISNIL(al_697))
      return (_704);
    else if (ISNIL(dl_698))
      return (_704);
    else if (HEAD(dl_698) != 0) {
      _704 = MIDCRA(HEAD(Ml_696), HEAD(ml_694), HEAD(mil_695), _704,
        HEAD(al_697));
      dl_698 = TAIL(dl_698);
      al_697 = TAIL(al_697);
      Ml_696 = TAIL(Ml_696);
      mil_695 = TAIL(mil_695);
      ml_694 = TAIL(ml_694);
    } else
      return (_704);
  return (_704);
}

static inline void loop_1025(LIST_387 yl_706, LIST_384 dk_716, LIST_384
  pk_717, IntegerList_99 Pk_718, LIST_384 ik_719, IntegerList_99 * _1026)
{
  while (!ISNIL(yl_706)) {
    _1026 = RCONS(loop_705(pk_717, ik_719, Pk_718, HEAD(yl_706), dk_716,
      DTOI(1)), _1026);
    yl_706 = TAIL(yl_706);
  }
  *_1026 = NIL;
  return;
}

static inline IntegerList_99 loop_709(LIST_387 yl_706, LIST_384 dk_716,
  LIST_384 pk_717, IntegerList_99 Pk_718, LIST_384 ik_719)
{
  IntegerList_99 _1026;
  loop_1025(yl_706, dk_716, pk_717, Pk_718, ik_719, &_1026);
  return (_1026);
}

static inline Integer loop_686(LIST_384 ml_675, LIST_384 mil_676,
  IntegerList_99 Ml_677, LIST_384 al_678, LIST_384 dl_679, Integer _685)
{
  while (!ISNIL(ml_675))
    if (ISNIL(mil_676))
      return (_685);
    else if (ISNIL(Ml_677))
      return (_685);
    else if (ISNIL(al_678))
      return (_685);
    else if (ISNIL(dl_679))
      return (_685);
    else if (HEAD(dl_679) != 0) {
      _685 = MIDCRA(HEAD(Ml_677), HEAD(ml_675), HEAD(mil_676), _685,
        HEAD(al_678));
      dl_679 = TAIL(dl_679);
      al_678 = TAIL(al_678);
      Ml_677 = TAIL(Ml_677);
```

```
      mil_676 = TAIL(mil_676);
      ml_675 = TAIL(ml_675);
    } else
      return (_685);
  return (_685);
}

static inline RatVector_103 linSolve_104(IntMatrix_102 A_303, IntegerList_99
  b_304, INT n_305, LIST_384 PLIST_306)
{
  while (TRUE) {
    LIST_384 _1029;
    LIST_387 _1030;
    LIST_384 _1031;
    IntegerList_99 _1032;
    LIST_384 _1033;
    LIST_384 _1034;
    loop_1007(PLIST_306, DTOI(1), A_303, b_304, n_305, PLIST_306,
      IPROD(DTOI(2), IPROD(IFACTL(n_305), IEXP(loop_657(A_303, loop_651(b_304,
       DTOI(0))), n_305))), &_1029, &_1030, &_1031, &_1032, &_1033, &_1034);
    if (loop_691(_1029)) {
      LIST_384 ik_991 = _1033;
      IntegerList_99 Pk_992 = _1032;
      LIST_384 pk_993 = _1031;
      return (loop_714(loop_709(_1030, _1029, pk_993, Pk_992, ik_991),
        loop_686(pk_993, ik_991, Pk_992, _1029, _1029, DTOI(1))));
    } else
      PLIST_306 = _1034;
  }
}

/*
 * global entry function
 */

RatVector_103 linSolve(IntMatrix_102 A, IntegerList_99 b, INT n)
{
  return (linSolve_104(A, b, n, PRIME));
}

/*
 * End of file
 */
```

D The Runtime System Interface

```
/*****************************************************************************
 *
 * dc.h                                                   Wolfgang Schreiner
 * runtime system interface of DC compiler
 *
 *****************************************************************************/

/* $Id: dc.h,v 1.6 1993/07/21 00:02:13 ws Exp ws $ */

#ifndef DC_H
#define DC_H

#if !defined(DC_INLINE) || !defined(__GNUC__)
#define __inline__
#endif

#ifdef DC_UNIX

/*****************************************************************************
 * simple Unix-based runtime system
```

```
 ****************************************************************************/

...omitted ...

#else

/****************************************************************************
 * SACLIB runtime system
 ****************************************************************************/

#define NO_SACLIB_MACROS
#include "saclib.h"
#include <stdio.h>

#define WORD  Word

#define INT   WORD
#define BOOL  WORD
#define POLY  WORD
#define NILT  WORD
#define SGNAL WORD

#define FALSE 0
#define TRUE  1

#define ERROR(s) do {fprintf(stderr, s); abort();} while (0)

__inline__
static Word _COMP(Word a, Word L) {
  WORD M;
  if (AVAIL == NIL) GC();
  M = AVAIL;
  AVAIL = RED(M);
  SFIRST(M,a);
  SRED(M,L);
  return(M);
}

__inline__
static Word *_RCOMP(Word a, Word *L) {
  WORD M;
  if (AVAIL == NIL) GC();
  M = AVAIL;
  AVAIL = RED(M);
  SFIRST(M,a);
  *L = M;
  return(SPACEB+M);
}

#define CONS(h,t)  _COMP(h,t)
#define RCONS(h,t) _RCOMP(h,t)
#define HEAD(l)    FIRST(l)
#define TAIL(l)    RED(l)

#define TUPLE2(a,b) \
  CONS(a, CONS(b, NIL))
#define TUPLE3(a,b,c) \
  CONS(a, CONS(b, CONS(c, NIL)))
#define TUPLE4(a,b,c,d) \
  CONS(a, CONS(b, CONS(c, CONS(d, NIL))))
#define TUPLE5(a,b,c,d,e) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, NIL)))))
#define TUPLE6(a,b,c,d,e,f) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, CONS(f, NIL))))))
#define TUPLE(l) l

#define ELEM1(l) \
  HEAD(l)
#define ELEM2(l) \
```

```
  HEAD(TAIL(l))
#define ELEM3(l) \
  HEAD(TAIL(TAIL(l)))
#define ELEM4(l) \
  HEAD(TAIL(TAIL(TAIL(l))))
#define ELEM5(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(l)))))
#define ELEM6(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(TAIL(l))))))
#define ELEMN(n,l) LELTI(l,n)

/* pseudo-function for integer to digit conversion, only if ISATOM(a)! */
__inline__
static Word ITOD(Word a) {
  return(a);
}

/* pseudo-function for digit to integer conversion */
__inline__
static Word DTOI(Word a) {
  return(a);
}

#endif

#endif

/*******************************************************************************
 * end of dc.h
 *******************************************************************************/
```