Entwurf und Implementierung einer erweiterbaren Schnittstelle zwischen *LooPo* und dem Haskell-Compiler GHC zur Anbindung wichtiger Datenstrukturen und Mathematik-Routinen von *LooPo*

(Programmierpraktikum am Lehrstuhl Prof. C. Lengauer, Ph. D.)

Armin Größlinger

Sommersemester 2001

Inhaltsverzeichnis

1 Einleitung

Das *LooPo*-Projekt [1] hat sich zum Ziel gesetzt, Methoden zur automatischen Schleifenparallelisierung mit Hilfe des Polyedermodells zu erforschen. Mittlerweile deckt *LooPo* alle wichtigen Phasen der Schleifenparallelisierung ab, z. B. Abhängigkeitsanalyse, Scheduler, Allokator, etc. Aufgrund des Modells sind dabei fortgeschrittene mathematische Methoden nötig, v. a. aus der Polyedertheorie.

Die Entwicklung von Parallelisierungsmethoden, die schließlich in einem Modul für *LooPo* implementiert werden sollen, erfordert häufig, viele Berechnungen im Polyedermodell durchzuführen, um die Eigenschaften der potenziellen neuen Methode zu untersuchen. Dabei ist es wünschenswert, dass diese Berechnungen interaktiv durchgeführt werden können, damit nicht für jede einzelne Berechnung ein eigenes Testprogramm geschrieben werden muss. Die Mathematikroutinen von *LooPo* sind in C bzw. C++ implementiert und daher recht „interaktionsfeindlich“.

Um die mathematischen Routinen interaktiv nutzen zu können, bieten sich mehrere Möglichkeiten:

(1) Computeralgebra-Systeme
Diese Systeme wurden für den Zweck einer interaktiven Benutzung entwickelt. Allerdings gibt es in diesen Systemen häufig nicht die in *LooPo* benötigten speziellen Methoden, sodass die erforderliche Funktionalität erst mit Hilfe der allgemeinen Funktionen des Algebra-Systems definiert werden müssten, was Codeverdoppelung und Fehleranfälligkeit bedeuten würde.

(2) Enwicklung einer interaktiven Schnittstelle zu *LooPo*
Eine interaktive Schnittstelle zu den Routinen von *LooPo* zu implementieren scheidet aufgrund der hohen Komplexität eines solchen Systems aus: Es wäre der Entwurf einer ganzen Sprache für diese Schnittstelle zusammen mit der Implementierung eines Interpreters dafür nötig.

(3) Anbindung von *LooPo* an eine interpretierte Hochsprache
Dieser Anzatz ähnelt (2), jedoch wird hier eine bestehende Sprache um einen Anschluss an *LooPo* erweitert, anstatt eine eigene Sprache zu entwickeln. Eine gegebene Sprache einzusetzen muss kein Nachteil gegenüber (2) sein, da – bei entsprechender Wahl der Hochsprache – die interaktive Schnittstelle von den Vorteilen und der Ausgereiftheit der verwendeten Sprache profitieren kann.

Die Lösung, die Mathematikroutinen von *LooPo* in eine bestehende, interpretierte Hochsprache einzubinden, erscheint insgesamt am effizientesten, da weder Codeverdoppelung noch der Entwurf einer ganzen Sprache erforderlich sind. Die Entscheidung über die Hochsprache ist auf Haskell gefallen, da es relativ einfach ist, C-Routinen in Haskell einzubinden und für Haskell sowohl Interpreter als auch Compiler existieren, sodass die Schnittstelle neben der interaktiven Nutzung im Interpreter auch die Implementierung von *LooPo*-Modulen in Haskell erlaubt.

2 Anforderungen an die Schnittstelle

Die Schnittstelle soll es ermöglichen, von Haskell aus die mathematischen Objekte, die in *LooPo* verwendet werden, zu manipulieren. Zu diesen zählen:

- Matrizen mit rationalen Einträgen,
- affine Funktionen und Ungleichungssysteme, die durch Matrizen repräsentiert werden,
- „höhere“ Datenstrukturen, z. B. Abhängigkeiten.

Die über die Schnittstelle anzuschließenden Operationen auf diesen Datenstrukturen sind

- Sortieren eines Ungleichungssystems im Sinne einer Fourier/Motzkin-Elimination (per *IneqTools* von *LooPo*),
- Berechnung der dualen Darstellung eines Ungleichungssystems (per PolyLib), sowie
- grundlegende Standardoperationen auf den Datenstrukturen, etwa Berechnung des Kerns einer Matrix, etc.

3 Entwurf der Schnittstelle

3.1 Repräsentation der Daten

Eine zentrale Entscheidung beim Entwurf der Schnittstelle ist die Festlegung, wo die Datenstrukturen verwaltet werden. Dabei gibt es die Möglichkeit, für jeden C/C++-Datentyp einen korrespondierenden Datentyp auf der Haskell-Seite zu unterhalten. Alternativ wäre es möglich, die Datenstrukturen ausschließlich auf der C-Seite zu haben und auf der Haskell-Seite lediglich Handles für die Datenobjekte zu halten. Aufgrund der Laziness und funktionalen Natur von Haskell würde dies aber voraussetzen, dass die Handles nur in einer Monade existieren könnten (die referentielle Transparenz könnte sonst nicht gewährleistet werden).

Aus diesem Grund erscheint es günstiger, eigene Datentypen auf der Haskell-Seite einzuführen. Als zusätzlichen Vorteil ermöglicht dieser Ansatz, die volle Funktionalität von Haskell für die Manipulation der Datenstrukturen auszunutzen.

3.2 Auswahl der C↔Haskell Schnittstelle

Für die Realisierung des Anschlusses der C/C++-Routinen an Haskell stehen mehrere Standard-Tools zur Verfügung. Alle Tools können nur Anbindungen von C-Funktionen realisieren, so dass für die in C++ implementierten Routinen in jedem Fall Wrapper in C geschrieben werden müssen.

- GreenCard [2]
  GreenCard ist ein Präprozessor, der aus einem mit speziellen Direktiven angereicherten Haskell-Source-File C- und Haskell-Quellcode für die Anbindung von C-Routinen an Haskell erzeugt. Dazu müssen für jeden Datentyp eine GreenCard-Direktive (die ggf. Haskell-Code enthält) und für zusammengesetzte Datentypen (für die GreenCard keine automatische Konvertierung vornehmen kann) C- (bzw. C++-) Funktionen geschrieben werden, die die Konvertierung vornehmen. Als großer Nachteil von GreenCard stellt sich die schwierige Speicherverwaltung heraus, so dass speziell bei C++-Funktionen, die einen zusammengesetzten Datentyp auf die Haskell-Seite übergeben, der Umweg über die Darstellung der Datenstruktur als Zeichenkette erforderlich ist.

- HaskellDirect [3]
  HaskellDirect kann Haskell-Datenobjekte über Schnittstellen, die mit der *Interface Description Language* (IDL) beschrieben sind, austauschen. HaskellDirect ist das mächtigste der hier vorgestellten Schnittstellen-Tools, da die Kommunikation dabei auch über COM erfolgt kann, allerdings kann HaskellDirect – wie

die anderen Schnittstellen-Tools – nur C und nicht C++ anbinden. Da *LooPo* keine in IDL beschriebene C-Schnittstelle besitzt, scheidet HaskellDirect aus.

- C→Haskell [4]
  C→Haskell kann aus C-Header-Files automatisch eine Schnittstelle zu den zugehörigen Funktionen und Datenstrukturen generieren. Da auch C++-Datentypen angeschlossen werden sollen, wäre bei diesem Ansatz neben Wrappern für die C++-Funktionen auch Wrapper-Datenstrukturen für die C++-Klassen erforderlich, in Gegensatz zu GreenCard und FFI.

- Foreign Function Interface (FFI) [5]
  Das Foreign Function Interface ist eine Erweiterung von Haskell, die es erlaubt, C-Funktionen relativ einfach an Haskell anzubinden. Das FFI kann einfache C-Datentypen zwischen C und Haskell transportieren (z. B. `int`). Insoweit ist das FFI mit GreenCard vergleichbar. Für zusammengesetzte Datentypen gibt es alldings im FFI keine Unterstützung.

Insgesamt scheint das FFI die beste Lösung für die Anbindung zu sein, da für die C++-Funktionen und zusammengesetzten Datentypen in jedem Fall eine spezielle Unterstützung auf der C-Seite implementiert werden muss.

3.3 Speicherverwaltung

Um Speicherlecks zu vermeiden, soll die Speicherverwaltung so weit wie möglich von der Haskell-Seite durch Aufruf entsprechender Allokations- und Deallokationsfunktionen übernommen werden. Diese Aufrufe lassen sich (siehe Abschnitt 4.2) auf der Haskell-Seite der Schnittstelle automatisieren.

Prinzipieller Ablauf eines Aufrufs $y = f(x_1, \ldots, x_k)$, wobei f auf der C/C++-Seite durch die Funktion $\hat{f}$ implementiert ist:

(1) Für alle $i \in \{1, \ldots, k\}$:

- Falls x_i ein zusammengesetztes Objekt ist:
  (a) alloziere des Speichers auf der C-Seite für x_i (an Adresse a_i)
  (b) übertrage x_i von der Haskell-Seite in den allozierten Speicher auf der C-Seite
- Falls x_i ein primitives Objekt ist: setze $a_i := c_i$.

(2) Rufe die C-Funktion $\hat{f}(a_1, \ldots, a_n)$ auf; erhalte das Ergebnis in z.

(3) Erzeuge aus z die entsprechende Haskell-Datenstruktur y.

(4) Falls z ein zusammengesetztes Objekt darstellt, dealloziere den zugehörigen Speicher.

(5) Für alle $i \in \{1, \ldots, k\}$: Gib a_i frei, falls es auf einen in (1) allozierten Speicherbereich zeigt.

Mit diesem Vorgehen werden zusammengesetzte Datenobjekte der C-Seite in Haskell stets durch eine Adresse, also einen Pointer, dargestellt. Dadurch kann die Haskell-Seite diese Objekte (als Pointer) an C-Funktionen übergeben, ohne ihren Aufbau kennen zu müssen. In der Implementierung zeigt sich dies dadurch, dass die Haskell-Repräsentationen der betreffenden Datenstrukturen opake Pointer (vom Typ `Ptr a`) sein werden.

4 Implementierung der Schnittstelle

Bei der Implementierung der Schnittstelle ergeben sich nun 3 Hauptaufgaben:

- Implementierung der *LooPo*-Datenstrukturen in Haskell,
- Implementierung der C↔Haskell Konvertierung der Datenstrukturen, sowie der automatischen Allokation/Deallokation in Haskell, und
- Implementierung von Wrapperfunktionen in C für die C++-Funktionen von *LooPo*, sowie von Hilfsfunktionen in C für die Allokation und Deallokation der C-seitigen Datenobjekte.

Die Realisierung dieser 3 Hauptaufgaben wird im folgenden an repräsentativ ausgewählten Beispielen für die Datenstrukturen bzw. Funktionen erläutert werden.

4.1 C-Seite

Die Aufgaben der C-Seite (wrappen der C++-Funktionen und (de-)allozieren von Speicher) sollen hier am Beispiel der Verwaltung der Datentypen `Fraction` und `FMatrix` gezeigt werden.

Bei `Fraction` sind 3 Funktionen nötig: Erzeugung (`newFraction_c`), Löschung (`deleteFraction_c`) und Extraktion von Zähler und Nenner (`decompFraction_c`). Die Haskell-Seite übergibt einen Bruch an `newFraction_c` bzw. erhält ihn von `decompFraction_c` als ein zweielementiges Array (von Typ `long long [2]`), wobei der erste Eintrag den Zähler des Bruches, der zweite den Nenner bezeichnet.

```
Fraction *newFraction_c(long long *p) {
    return new Fraction(p[0], p[1]);
}

void deleteFraction_c(Fraction *f) {
    delete f;
}

long long *decompFraction_c(Fraction *f) {
    long long *p = (long long *)malloc(2 * sizeof(long long));
    p[0] = f->get_num();
    p[1] = f->get_denom();
    return p;
}
```

Der Anschluss des Datentyps `FMatrix` funktioniert völlig analog. Die Konstruktion und die Zerlegung der Matrix erfolgen mit Hilfe eines Arrays, das die Einträge der Matrix zeilenweise speichert, d. h. ein Array von Pointern auf `Fraction` ist.

```
FMatrix *newFMatrix_c(int rows, int cols, Fraction **coeffs) {
    FMatrix *fm = new FMatrix(rows, cols);
    for (int r=0; r<rows; r++) {
        for (int c=0; c<cols; c++)
            fm->set_entry(r, c, **coeffs++);
    }
    return fm;
}

void deleteFMatrix_c(FMatrix *fm) {
    delete fm;
}

int rowsFMatrix_c(const FMatrix *fm) {
    return fm->get_rows();
}

int colsFMatrix_c(const FMatrix *fm) {
    return fm->get_cols();
}

Fraction **entriesFMatrix_c(const FMatrix *fm) {
    const int rows = fm->get_rows();
    const int cols = fm->get_cols();
    Fraction **entries = (Fraction **)malloc(sizeof(Fraction *)*rows*cols);
    Fraction **p = entries;
    for (int r=0; r<rows; r++) {
        for (int c=0; c<cols; c++)
            *p++ = new Fraction(fm->get_entry(r,c));
    }
    return entries;
}
```

4.2 Haskell-Seite

Auf der Haskell-Seite gibt es 2 Dinge zu erledigen: Datenstrukturen zu definieren und die Konvertierung der Datenstrukturen zwischen C und Haskell vorzunehmen.

4.2.1 Datenstrukturen

Die Definition der Datenstrukturen stellt keine große Herausforderung dar, da die Struktur im Wesentlichen der Struktur der Datenstrukturen auf der C++-Seite entspricht. Für unsere Beispiele, die C++-Datenstrukturen `Fraction` und `FMatrix`, verwenden wir unter Haskell den vorgegebenen Typ `Rational` bzw. den Datentyp `Matrix`,

```
data Matrix = Matrix { coeffs :: Array (Integer,Integer) Rational }
              deriving (Eq)
```

der einfach einem 2-dimensionalem Feld dessen Einträge `Rationals` sind, entspricht.

4.2.2 Konvertierung

Der Aufruf einer Funktion der C-Seite wird nach Abschnitt 3.3 immer nach dem gleichen Muster durchgefürt: Zuerst werden die Argumente in die C-Darstellung konvertiert, dann die C-Funktion aufgerufen, dann das Ergebnis in eine Haskell-Datenstruktur umgewandelt und schließlich wird ggf. der Speicher für Argumente und Ergebnis freigegeben.

Um diese Abfolge von Aktionen zu automatisieren und Fehler in der Speicherverwaltung zu vermeiden, bietet es sich an, diesen Ablauf durch Funktionen zu beschreiben, die über die Typen der beteiligten Datenstrukturen in Haskell und C parametrisiert sind:

```
class CConv a b where
    toC   :: a -> (b -> IO c) -> IO c
    fromC :: b -> IO a
```

Die Typvariable `a` entspricht dabei dem Typ der Haskell-Seite und `b` dem (Pseudo-)Typ (siehe unten) der C-Seite. `fromC` ist – wie der Name andeutet – dafür zuständig, ein von einer C-Funktion geliefertes Ergebnis z (vom Typ `b`) in eine Haskell-Datenstruktur vom Typ `a` umzuwandeln und den Speicher, der von z belegt wird, freizugeben. Das Gegenstück zu `fromC` ist `toC`, welches für die Umwandlung einer Haskell-Datenstruktur in die C-Darstellung zuständig ist. `toC` hat aber im Gegensatz zu `fromC` zwei Speicherverwaltungsaufgaben zu erledigen: die Allokation des Speichers für die Datenstruktur *vor* dem Aufruf der C Funktion und die Deallokation dieses Speicherbereichs *nach* dem erfolgten Aufruf. Dies erklärt die kompliziertere Signatur von `toC`, welches die zu konvertierende Datenstruktur (vom Typ `a`) und eine Fortsetzungsfunktion f (vom Typ `b -> IO c`) übergeben bekommt. `toC` konvertiert dann die Datenstruktur in die C-Darstellung (mit Typ `b`) und übergibt sie an die Fortsetzungsfunktion. Als Ergebnis liefert `toC` das Ergebnis von f, gibt aber nach Abarbeitung von f den vorher belegten Speicher frei.

Damit kann der Anschluss einer C-Funktion (hier: Determinante einer Matrix) im Prinzip wie folgt erfolgen:

```
foreign import determinant_c :: Ptr Matrix -> IO (Ptr Rational)
determinant :: Matrix -> IO Rational
determinant matr = toC matr (\matr_c ->
                             determinant_c matr_c >>= fromC)
```

Der Interface-Code, d. h. die `toC`s für die Argumente einer Funktion f und das abschließende `fromC` für den Rückgabewert nehmen stets folgende Form an:

$$\texttt{toC}\,x_1\,\Big(\lambda y_1 \to \texttt{toC}\,x_2\,(\lambda y_2 \to \cdots \texttt{toC}\,x_n\,(\lambda y_n \to f\,x_1\,x_2\,\cdots\,x_n \texttt{ >>= fromC})\cdots)\Big)$$

Es bietet sich daher an, diese Folge von `toC`s durch Funktionen zu realisieren (hier beispielhaft für ein und zwei Argumente; die Implementierung enthält alle angegebenen Funktionen in Varianten für bis zu vier Argumente):

```
cppiArgs1 :: (CConv a a') => (a' -> IO b) -> (a -> IO b)
cppiArgs1 f x = toC x (\x' -> f x')

cppiArgs2 :: (CConv a a', CConv b b') =>
             (a' -> b' -> IO c) -> (a -> b -> IO c)
cppiArgs2 f x y = toC x (\x' -> cppiArgs1 (f x') y)
```

Für die Konvertierung der Argumente und des Rückgabewerts können folgende Funktionen definiert werden:

```
cppiConvert1 :: (CConv a a', CConv b b') =>
                (a' -> IO b') -> (a -> IO b)
cppiConvert1 f x = cppiArgs1 f x >>= fromC

cppiConvert2 :: (CConv a a', CConv b b', CConv c c') =>
                (a' -> b' -> IO c') -> (a -> b -> IO c)
cppiConvert2 f x y = cppiArgs2 f x y >>= fromC
```

Obiges Beispiel kann damit einfacher als

```
determinant :: Matrix -> IO Rational
determinant = cppiConvert1 determinant_c
```

realisiert werden. Unschön ist an der bisher entwickelten Schnittstelle, dass (entsprechend der Definition der Klasse `CConv`) alle angebundenen Funktionen in der `IO` Monade operieren, um die Allokation und Deallokation des Speichers durchführen zu können. Die meisten der anzuschließenden Funktionen f besitzen aber neben den Speicher-(De-)Allokationen keine Seiteneffekte, so dass eine vollständige Ausführung von `cppiConvert`N f die referentielle Transparenz nicht verletzt, selbst wenn die Auswertung des Ausdrucks nicht durch die `IO` Monade sequenzialisiert ist. Für diesen Fall stellt die Schnittstelle auch „unsichere“ Varianten von `cppiConvert`N bereit:

```
unsafeCppiConvert1 :: (CConv a a′, CConv b b′) =>
                      (a′ -> IO b′) -> (a -> b)
unsafeCppiConvert1 f x = unsafePerformIO ((cppiConvert1 f) x)

unsafeCppiConvert2 :: (CConv a a′, CConv b b′, CConv c c′) =>
                      (a′ -> b′ -> IO c′) -> (a -> b -> c)
unsafeCppiConvert2 f x y = unsafePerformIO ((cppiConvert2 f) x y)
```

Die tatsächlich Realisierung von `determinant` ist damit durch

```
determinant :: Matrix -> Rational
determinant = unsafeCppiConvert1 determinant_c
```

gegeben und erfordert nicht den Einsatz der `IO` Monade.

Es ist nun noch die Frage zu beantworten, welche Typen `b` in den Instanzen der Klasse `CConv` als C-Pendants der Haskell-Datentypen stehen sollen: es muss sich dabei natürlich jeweils um einen korrekten Haskell-Typ handeln, der aber andererseits vom FFI beim Aufruf einer C-Funktion verwendet werden kann. Das FFI stellt einige Typen in den Modulen `CTypes` und `Foreign` zur Verfügung, von denen wir nur `CInt`, `CLLong` und `Ptr a` verwenden werden. `CInt` entspricht dem C-Typ `int` und wird für die Übergabe von Ganzzahlen (Typen der Klasse `Integral`) verwendet. Zähler und Nenner von `Rationals` werden (wegen des größeren Zahlenbereichs) als `CLLong` übergeben, was `long long` in C entspricht. Alle anderen Typen *X* der Haskell-Seite werden durch den Typ `Ptr` *X* dargestellt. Dabei ist zu beachten, dass der Typ `Ptr` *X* unabhängig von der Wahl des Typs *X* stets einem C Adresszeiger entspricht. Das Typargument des Typkonstruktors `Ptr` ist also ein Phantomtyp, d. h. es dient lediglich dazu, für das Typsystem unterschiedliche Typen zu schaffen, obwohl die Repräsentation in jedem Fall dieselbe ist.

Dies erklärt nun die Typen `Ptr Matrix` und `Ptr Rational` im obigen Beispiel, die lediglich dazu dienen, die passende Instanz der Typklasse `CConv` für die Aufrufe von `toC` und `fromC` zu wählen.

Die Typen der Klasse `Integral` werden, wie oben erwähnt, auf der C-Seite durch den Typ `int` dargestellt. Da hierfür keine (De-)Allokationen erforderlich sind, sieht die entsprechende Instanz von `CConv` so aus (`CInt` ist selbst Instanz von `Integral`, deshalb kann die Konvertierung einfach durch `fromIntegral` durchgeführt werden):

```
instance Integral a => CConv a CInt where
    toC   x f = f (fromIntegral x)
    fromC x   = return (fromIntegral x)
```

Für `Rational` ist etwas mehr zu tun, da hier explizite Speicherverwaltung erforderlich ist. Wir erhalten mit den früher gegebenen Definitionen der Hilfsfunktionen auf der C-Seite:

```
foreign import newFraction_c :: Ptr CLLong -> IO (Ptr Rational)
foreign import decompFraction_c :: Ptr Rational -> IO (Ptr CLLong)
foreign import deleteFraction_c :: Ptr Rational -> IO ()
```

```
instance CConv Rational (Ptr Rational) where
    toC rat action =
        do frac <- withArray [num, denom] newFraction_c
           res <- action frac
           deleteFraction_c frac
           return res
        where
        num   = fromIntegral $ numerator rat
        denom = fromIntegral $ denominator rat
    fromC frac =
        do p <- decompFraction_c frac
           [num, denom] <- peekArray 2 p
           free p
           return ((fromIntegral num) % (fromIntegral denom))
```

Der Anschluss von `Matrix` funktioniert entsprechend; hier zeigt sich die Nützlichkeit der oben definierten Funktionen `cppiArgs`N, da z. B. die Argumente der Funktion `newFMatrix_c` auch von Haskell nach C konvertiert werden müssen.

```
foreign import newFMatrix_c :: CInt -> CInt -> Ptr (Ptr Rational) ->
                               IO (Ptr Matrix)
foreign import deleteFMatrix_c :: Ptr Matrix -> IO ()
foreign import rowsFMatrix_c :: Ptr Matrix -> IO CInt
foreign import colsFMatrix_c :: Ptr Matrix -> IO CInt
foreign import entriesFMatrix_c :: Ptr Matrix -> IO (Ptr (Ptr Rational))

instance CConv Matrix (Ptr Matrix) where
    toC matr action =
        do fm <- (cppiArgs3 newFMatrix_c) (nRows matr) (nColumns matr)
                                          (entries matr)
           res <- action fm
           deleteFMatrix_c fm
           return res
    fromC fm =
        do rs <- rowsFMatrix_c fm
           cs <- colsFMatrix_c fm
           coeffs <- entriesFMatrix_c fm
           entries <- peekArray (fromIntegral (rs*cs)) coeffs >>= \xs ->
                      sequence (map fromC xs)
           free coeffs
           deleteFMatrix_c fm
           return $ matrix (rs, cs, entries :: [Rational])
```

4.3 Zusammenfassung der Schnittstelle

Die Schnittstelle bietet für folgende Typen einen Anschluss zwischen Haskell und C.

Haskell	C
`Integral a => a`	`int`
`Bool`	`int`
`Enum a => a`	`int`
`Integral a => [a]`	`int *`
`Rational`	`Fraction *`
`Matrix`	`FMatrix *`
`IneqSystem`	`FIneqSystem *`

Zusätzlich ist noch folgende Instanz definiert, mit deren Hilfe jeder Typ `Maybe` X als Pointer auf der C-Seite dargestellt wird, wenn X selbst auf der C-Seite als Adresszeiger repräsentiert wird. `Nothing` wird dabei auf den `NULL` Pointer abgebildet.

```
instance CConv a (Ptr b) => CConv (Maybe a) (Ptr b) where
    toC Nothing  f = f nullPtr
    toC (Just x) f = toC x f

    fromC y
        | y == nullPtr  =  return Nothing
        | otherwise     =  fromC y >>= \x -> return (Just x)
```

Folgende Funktionen sind mit Hilfe der Schnittstelle durch ihre Pendants auf der C-Seite implementiert:

```
determinant :: Matrix -> Rational
kernel :: Matrix -> Matrix
gaussJordan :: Matrix -> Matrix -> Maybe (Matrix, [Integer])
echelonReduction :: Matrix -> (Matrix, Matrix)
inverseMatrix :: Matrix -> Matrix
sortIneqs :: IneqSystem -> ToolMethod -> Maybe IneqSystem
eliminateRedundant :: IneqSystem -> Maybe IneqSystem ->
                      ToolMethod -> IneqSystem
isSuperset :: IneqSystem -> IneqSystem -> Bool
dualOfSystem :: EqSystem -> IneqSystem -> LRV
```

Literatur

[1] http://www.infosun.fmi.uni-passau.de/~loopo/

[2] http://haskell.org/greencard/

[3] http://haskell.org/hdirect/

[4] http://www.cse.unsw.edu.au/~chak/haskell/c2hs/

[5] http://www.cse.unsw.edu.au/~chak/haskell/ffi/