Übersetzung von Curry in imperative strikte Sprachen am Beispiel von C

Marc Grauel

Betreuer: Prof. Dr. Michael Hanus, Dipl. Inf. Klaus Höppner

21. Februar 2006

Inhaltsverzeichnis

Zusammenfassung

Diese Studienarbeit beschreibt Strategien zur Übersetzung der Multiparadigmenspache Curry, die logische und funktionale Programmierung vereint, in strikte Hochsprachen am Beispiel von C. Die Übersetzung wird möglichst direkt und ohne Modellierung einer abstrakten Maschine durchgeführt, wodurch allerdings eine Unterstützung bestimmter Fähigkeiten von Curry entfällt. Abschließend soll erörtert werden, wie sich der erarbeitete Ansatz zur Übersetzung auf den Ressourcenbedarf der erzeugten Programme auswirkt. Hierzu findet ein Vergleich mit anderen Curry-Compilern statt.

1 Einführung

Die Programmiersprache Curry verbindet funktionale und logische Programmierung miteinander und stellt so eine geeignete Plattform zur Erforschung, Lehre und Anwendung logisch funktionaler Sprachen dar. Um von den Vorteilen von Curry als deklarative Hochsprache auch bei der Entwicklung von Software für geschwindigkeitskritische Anwendungen oder Systemen mit eingeschränkter Architektur zu profitieren, kann es notwendig werden, Curry-Programme in andere Sprachen zu überführen. Denkbar ist zum Beispiel der Einsatz von Curry bei der Programmierung von eingebetteten Systemen, wofür eigens eine spezialisierte Curry-Variante namens „Embedded Curry“ entwickelt wurde. Bei der Übersetzung von Embedded-Curry-Programmen wird auf die verbreitete Technik zurückgegriffen, C als plattformunabhängige Assemblersprache zu nutzen. [3][8] Ein anderes Anwendungsgebiet für Curry stellt die Erzeugung dynamischer Webinhalte dar. Wie z.B. in [6] beschrieben eignet sich Curry gut zur Entwicklung komplexer Webanwendungen, allerdings stellen die Größe und Performanz der erzeugten Programme beim Einsatz auf einem Webserver möglicherweise ein Problem dar. Daher wäre es in diesem Fall evtl. sinnvoll, die in Curry entwickelten Programme in spezialisierte Sprachen wie PHP zu übersetzen. Auch in diesem Fall wäre wieder ein Überführen in C zur Erzeugung von CGI-Programmen denkbar. Gemein ist den möglichen Zwischensprachen allerdings, dass sie im Gegensatz zu Curry strikt sind. Curry wertet Ausdrücke lazy, d.h. verzögert und bei Bedarf, aus und folgt damit einer anderen Strategie als strikte Sprachen z.B. C++ oder Java. Die direkte Übersetzung von Curry-Programmen in strikte Sprachen soll im Rahmen dieser Arbeit näher betrachtet werden, dabei soll nicht etwa wie in ähnlichen Projekten (z.B. dem Münster Curry Compiler (MCC) [5]) eine abstrakte Maschine in der Zielsprache modelliert werden. Vielmehr soll soweit möglich versucht werden Datenstrukturen und Funktionen direkt in die Zielsprache zu überführen, um so einen Einsatz auf Plattformen mit stark eingeschränkten Ressourcen zu ermöglichen. Logische Funktionalitäten, z.B. Constraint Solving, und die verzögerte Auswertung von Curry werden bei solch einer Vorgehensweise naturgemäß nicht unterstützt. Exemplarisch soll im folgenden ein Ansatz zur Übersetzung von Curry-Programmen in C-Programme näher betrachtet werden. Eine Übersetzung in andere imperative Sprachen würde aber grob einem vergleichbaren Schema folgen. Die Wahl von

C als Übersetzungsziel hat zudem für diese Arbeit den Vorteil, dass auf Erfahrungen und Ansätze zurückgegriffen werden kann, die bei der Entwicklung von Embedded-Curry gemacht wurden[2]. Embedded-Curry ist zur Programmierung von Lego-Mindstorms entworfen worden und beschränkt sich ebenfalls nur auf einen Teil des Funktionsumfangs von Curry, da eine recht direkte Übersetzung in C-Code durchgeführt wird. Allerdings sind die Möglichkeiten von Embedded-Curry bis dato eingeschränkt, weil mit Rücksicht auf die Zielplattform keine Speicherverwaltung zur Verfügung steht, und somit eine Möglichkeit dynamische und rekursiv definierte Datentypen zu verwenden fehlt. Das Überführen von Curry in C ermöglicht des Weiteren eine hohe Portabilität des Codes, da C-Compiler für nahezu alle gängigen Rechnerarchitekturen existieren. Die Entwicklung eines Übersetzers für Hochsprachen wie Curry wird dadurch wesentlich einfacher, weil kein plattformabhängiger Assemblercode erzeugt werden muss; C bietet hierbei trotzdem einen hohen Grad an Hardwarenähe. Ferner kann durch das Übersetzen nach C von Optimierungen profitiert werden, welche von aktuellen C-Compilern beherrscht werden. Auch die große Menge an für C existierenden Werkzeugen, Libraries und Dokumenten, sowie der hohe Bekanntheitsgrad sprechen für eine Wahl von C als „Zwischensprache“ (intermediate language). Problematisch an der Verwendung von C ist der Mangel eines integrierten Mechanismus zur Speicherbereinigung (garbage collection), allerdings kann dieses Problem leicht mit der Verwendung externer Programmbibliotheken zur Speicherbereinigung behoben werden. Prinzipiell wäre eine Übersetzung der Curry-Programme in portable Assemblersprachen wie `C--` [7] denkbar, die eine Garbage-Collection bereitstellen. Diese Möglichkeiten sollen hier aber unberücksichtigt bleiben, da es plattformunabhängigen Assemblersprachen an der nötigen Verbreitung fehlt, und diese Studienarbeit prinzipiell als Zielsetzung eine experimentelle Übersetzung von Curry in eine strikte Hochsprache wie C hat. Die im Rahmen dieser Arbeit entstandenen Übersetzungstechniken und gesammelten Erfahrungen sollen im Folgenden geschildert werden, wobei abschließend ein Vergleich des Ressourcenbedarfs des erzeugten Codes mit dem anderer Curry-Compiler durchgeführt werden soll. Zunächst wird aber eine kurze Einführung in die Sprachen Curry und Flat-Curry gegeben werden, dann wird die Übersetzung von Curry in C geschildert, wobei auf Speicherverwaltung, Modellierung von Datentypen und Funktionen eingegangen wird. Am Beispiel eines konkreten Programmes werden alle vorgestellten Übersetzungsstrategien noch einmal im Ganzen aufgezeigt ,und im Anschluss daran werden Einschränkungen und mögliche Optimierungen angesprochen. Nach der Vorstellung des Compilerprototyps, wird dieser mittels Benchmarks mit anderen Curry-Compilern verglichen werden.

2 Curry

Im Folgenden soll ein grober Überblick über Curry gegeben werden; dies erfolgt nur zum besseren Verständnis späterer Absätze und stellt keine vollständige Einführung dar. Detailfragen sollten in [1] nachgeschlagen werden.

Curry vereint mit funktionaler und logischer Programmierung die wichtigsten Paradigmen deklarativer Programmiersprachen. So beherrscht Curry Elemente funktionaler (z.B. Funktionen höherer Ordnung oder die verzögerte Auswertung von Ausdrücken (lazy evaluation)) und logischer Programmiersprachen (z.B. logische Variablen, partielle Datenstrukturen). Ferner bietet Curry die Möglichkeit zur nebenläufigen Programmierung (nebenläufige Auswertung von Constraints mit Synchronisation durch logische Variablen). Curry-Programme sind aus syntaktischer Sicht funktionale Programme, die um freie Variablen erweitert werden können. Wie schon in der Einleitung erwähnt, sind die Fähigkeiten Currys zur logischen und nebenläufigen Programmierung für diese Arbeit nicht von Bedeutung, und daher sind im folgenden mit Curry-Programmen solche gemeint, die nur die Aspekte der rein funktionalen Programmierung nutzen.

Zunächst soll der Aufbau von Curry-Programmen genauer betrachtet werden. Curry-Programme bestehen aus der Definition von Datentypen und Funktionen. Die Deklaration von Datentypen beginnt mit dem Schlüsselwort `data` gefolgt vom Namen des Datentyp und den Definitionen der Konstruktoren des Typs.

```
data Bool        = True | False
data Complex     = Compl Float Float
```

Das Beispiel zeigt die Deklaration eines Datentypen `Bool`, der dem vordefinierten in Curry entspricht, und eines Datentypen, welcher zur Modellierung von komplexen Zahlen dienen könnte. `True` und `False` sind Konstanten bzw. nullstellige Konstruktoren und `Compl` ist der Name des Konstruktors für komplexe Zahlen, der als Parameter zwei Fließkommazahlen fordert.

Auf ähnliche Weise können so auch rekursive Datentypen definiert werden, wie es im folgenden Beispiel für eine Liste ganzer Zahlen geschehen soll.

```
data IntListe    = ElementI Int IntListe | LeereIntListe
```

Wie leicht zu erkennen ist, besteht ein Listenelement aus einer Zahl vom Typ `Int` und einer weiteren Teilliste vom Typ `IntListe`. Es lässt sich auch ein allgemeiner Listentyp definieren, welcher Listen beliebiger Typen ermöglicht, was im nächsten Beispiel aufgezeigt werden soll. Die Definition erfolgt analog zum Beispiel der Liste ganzer Zahlen. Man beachte aber, dass nach dem Namen des Datentypen jetzt eine Typvariable `a` eingeführt wird, welche für einen nicht näher spezifizierten Typ steht.

```
data Liste a     = Element a Liste | LeereListe
```

Die Deklaration von Funktionen beginnt in Curry mit dem nicht zwingend notwendigen aber hilfreichen Festlegen der Eingabetypen und des Ergebnistyps,

was durch :: nach dem Funktionsnamen gekennzeichnet wird. Das folgende Beispiel zeigt eine Funktion zur Bestimmung der Länge einer Liste.

```
listLength :: Liste a -> Int
listLength LeereListe            = 0
listLength (Element x restliste) = 1 + (listLength restliste)
```

Ein interessanter Aspekt von Curry sind Funktionen höherer Ordnung. Funktionen höherer Ordnung sind Funktionen, die als Eingabe eine Funktion erhalten oder als Ergebnis eine Funktion liefern. Eine typische Anwendung für solche Abbildungen ist zum Beispiel der Befehl `map` des Prelude-Moduls, welcher eine übergebene Funktion auf alle Elemente einer Liste anwendet.

```
map             :: (a->b) -> [a] -> [b]
map _ []        = []
map f (x:xs)    = f x : map f xs
```

2.1 Flat-Curry

Mit Flat-Curry wurde ein Zwischensprache geschaffen, welche als Schnittstelle zwischen verschiedenen Applikationen dient, die Curry-Programme verarbeiten. Flat-Curry bringt einige Vereinfachungen mit sich, so existieren etwa keine lokalen Funktionsdeklarationen, daher werden bei der Übersetzung in Flat-Curry lokal deklarierte Funktionen mittels Lambda-Liftings auf die globale Ebene überführt. Des Weiteren wird das Pattern-Matching nicht wie in Curry durch die mehrfache Deklaration einer Funktion sondern explizit durch Case-Ausdrücke dargestellt. Aus der oben definierten `ListLength` wird in Flat-Curry-Darstellung z.B.:

```
listLength :: Liste a -> Int
listLength v0 = case v0 of
                LeereListe -> 0
                (Element v1 v2) -> 1 + listLength v2
```

Für Curry existiert ein Modul zur Darstellung von Flat-Curry-Programmen, um die Meta-Programmierung mit Curry zu unterstützen. Somit erleichtert Flat-Curry die Entwicklung eines Curry-Compilers, der selber auf Curry basiert. Der im folgenden beschriebene Compiler-Prototyp greift daher auch auf die Nutzung von Flat-Curry zurück und kann so von den syntaktischen Vereinfachungen profitieren.

3 Übersetzung Curry in C

Die in dieser Arbeit vorgestellten Ansätze zur Übersetzung von in Curry entwickelten Programmen in C-Programme basieren teilweise auf den Konzepten aus [2] und stellen eine Erweiterung dieser Arbeit dar. Allerdings ist die Zielplattform des C-Codes im Gegensatz zur erwähnten Arbeit nicht speziell auf Embedded Systems wie Lego Mindstorms festgelegt, sondern es wurde auch auf die Programmerzeugung auf diversen Systemen, z.B. IBM-kompatible PCs oder Sun Sparc Stations abgezielt. Ferner war es die Hauptabsicht, die Konzepte des existierenden Compilers so zu erweitern, dass z.B. auch rekursive und polymorphe Datentypen übersetzt werden können. Der neue entwickelte Compiler wird ebenfalls nur eine Teilmenge aller Curry-Programme kompilieren können. In abschließenden Benchmarks soll untersucht werden, ob diese Einschränkungen einen Vorteil darstellen.

Im folgenden sollen zwei Ansätze zur Übersetzung von Curry-Programmen aufgezeigt werden, die im Rahmen dieser Arbeit entstanden sind. Beiden Ansätzen ist gemein, dass die Funktionen des zu übersetzenden Curry-Programms soweit möglich und sinnvoll direkt in entsprechende C-Funktionen überführt werden. Diese direkte Art des Übersetzens ist wie schon angedeutet nur für Programme möglich, die auf die logischen Komponenten von Curry verzichten. Die erzeugten C-Programme werden sprachbedingt nur strikt ausgewertet, es findet also keine verzögerte Auswertung (lazy evaluation) statt.

3.1 Speicherverwaltung

In C steht standardmäßig keine automatische Speicherbereinigung (garbage collection) zur Verfügung, was bei der Übersetzung von Curry-Programmen das Problem aufwirft, den geeigneten Zeitpunkt zur Freigabe von nicht mehr benötigtem Speicher, der zum Beispiel für dynamische Datentypen verwendet wurde, zu bestimmen. In der Implementierung des Curry2C-Compilers wird daher auf den Boehm-Garbage-Collector zurückgegriffen, der die notwendigen Speicherbereinigungen vornimmt. Auf den Boehm-Garbage-Collector wurde aus einer Vielzahl von Gründen zurückgegriffen: Zum einen lässt er sich nahezu transparent in bestehende C-Programme integrieren, zum anderen ist das System für viele Plattformen als C-Bibliothek frei verfügbar und hat sich in einer Vielzahl von Softwareprojekten bewährt. [4]

Ein Vorteil des gewählten Garbage-Collector-Systems ist die leichte Anwendung, da keine explizite Kooperation des Programms mit der Speicherverwaltung notwendig ist. Eine Berücksichtigung von besonderen Mechanismen, zum Beispiel Referenzzählern, die bei der Verwendung von Zeigerarithmetik nötig werden, entfällt hier völlig. Das Einbinden des Speicherbereinigers erfolgt durch Ersetzen des Speicherallozierungsbefehls `malloc()` und ist damit praktisch in jedem C-Programm möglich. Somit muß beim Generieren des C-Codes aus Curry-Programmen kein weiterer Aufwand für die Speicherverwaltung betrieben werden, was zur Einfachheit des Systems beiträgt.

3.2 Modellierung von Curry-Datentypen in C

Der erste naive Ansatz zur Übersetzung von dynamische Datenstrukturen verfolgt eine sehr direkte Strategie. Zunächst wird eine Liste von Enumeratoren erzeugt, die sich aus den Namen der Konstruktoren ergeben und zur Identifizierung dienen. Für jeden Konstruktor eines Datentyps wird ein C-Datentyp generiert, welcher aus einem Strukturdatentyp besteht, der alle Elemente des zu übersetzenden Unterdatentypen bündelt. Im Falle einer Liste ganzer Zahlen wird also eine C-Struktur für die Elemente und eine weitere für die leere Liste erzeugt. Die Struktur zur Elementdarstellung enthält dann eine Variable vom Typ `Int` und notwendigerweise eine Variable, die auf die Restliste mittels eines Zeigers verweist. Aufgrund der Deklarationsreihenfolge der Strukturen kann der Zeiger auf die Restliste nicht vom Listentyp sein und muss daher typenlos (`void`) deklariert werden, was bei der Übersetzung der Programme die Erzeugung geeigneter Typecasts erfordert, wenn auf die Elemente der Strukturen zugegriffen wird. In diesem Beispiel bedeutet dies, dass der Zeiger auf eine Liste etwa beim Anhängen an eine Liste in einen Void-Pointer umgewandelt werden muss. Umgekehrt sind auch Typumwandlungen der Zeiger beim Zugriff auf die Restliste notwendig. Prinzipiell sind solche Typumwandlungen nicht unproblematisch, da keine saubere Typüberprüfung möglich ist und somit es leicht zu Fehlern kommen kann. Im Falle der Übersetzung von Curry-Programmen stellen die Typumwandlungen aber kein Problem dar, da das zu übersetzenden Programm schon bei der Überführung in das Flat-Curry-Format vollständig auf Typkorrektheit überprüft wurde. Bei der Übersetzung nach C können also die beschrieben Umwandlungen gefahrlos durchgeführt werden. Mittels Definition eines weiteren C-Datentyps werden Strukturen aller Unterdatentypen eines Curry-Datentyps zusammengefasst, wofür eine Union-Struktur verwendet wird. Eine Variable vom Typ dieser Union-Struktur und eine Variable vom Typ des oben beschriebenen Aufzählungtyps bilden, in einer weiteren Struktur zusammengefasst, die Repräsentation eines Curry-Datentyps in C. Am Beispiel der Liste ganzer Zahlen sieht die beschriebene Übersetzung wie folgt aus:

```
data IntListe    = ElementI Int IntListe | LeereIntListe
```

Es ergibt sich folgender C-Quelltext falls das Curry-Modul `"‘test.curry"’` heißt:

```
enum test_IntListe_enum{e_test_ElementI,e_test_LeereIntListe};

struct sub_test_ElementI
{
  int   v0;      /* Listenelement */
  void* v1;      /* Zeiger auf Restliste */
};

struct sub_test_LeereIntListe
{
```

```
            /* Leere Dummystruktur */
};

union test_IntListe_union
{
  sub_test_ElementI       v_ElementI;
  sub_test_LeereIntListe  v_LeereIntListe;
};

struct test_IntListe
{
  test_IntListe_enum  _constr;   /* Konstruktoraufzählung */
  test_IntListe_union _union;    /* Parameter */
};
```

Zusätzlich wird für jeden Konstruktor eine C-Funktion erzeugt, welche die nötigen Initialisierungen, also zum Beispiel das Alloziieren des Speichers und Füllen der Datenstrukturen, durchführen.

Der obige Ansatz stellt allerdings noch keine Lösung zur Implementierung von polymorphen Datentypen zur Verfügung. Es müsste hierfür ein weiterer Datentyp zur Modellierung von Typvariablen eingeführt werden, in dem Typidenfikator und Zeiger auf Datum gekapselt werden müssten. Solch eine Struktur hätte z.B. folgendes Aussehen:

```
struct tvar
{
  enum prelude_tvar_enum  typeID; /* globaler Typidentifikator */
  void * data; /* typenloser Zeiger auf Daten */
};
```

Hieraus ergibt sich offensichtlich die Notwendigkeit einer globalen Typaufzählung oder Typidentifikatorliste. Auf den ersten Blick scheint so einer Realisierung von polymorphen Datentypen nichts mehr im Wege zu stehen. Analog zum Listenbeispiel könnte die Datenstruktur eines polymorphen Listenelements wie folgt implementiert werden:

```
struct sub_test_Element
{
  tvar* v0;      /* Listenelement */
  void* v1;      /* Zeiger auf Restliste */
};
```

Bei genauerer Betrachtung ergeben sich bei der beschrieben Lösung aber weitere Probleme. Die Realisierung des Gleichheitsoperators gestaltet sich bei solch einem Ansatz recht kompliziert, da für alle Datentypen Vergleichsfunktionen generiert werden müssen, welche elementweise die Gleichheit der Datenstrukturen überprüfen. Ähnliche Komplikationen treten auf, will man die Funktion `show` der Prelude implementieren. `show` stellt jeglichen Curry-Datentyp in einer

Zeichenkette dar und ist somit essentiell für die Ausgabe von Daten. Es müssten also vom Compiler für jeden Datentyp Funktionen erzeugt werden, die die Daten in einen String überführen. Prinzipiell wären solche Lösungen denkbar, allerdings zeigt sich an den auftretenden Problemen, dass eine nahezu direkte Übersetzung von Curry- in C-Datentypen eine Vielzahl aufwendiger Detaillösungen verlangt. Insbesondere verkomplizieren die zu genierenden Typecasts, Zugriffsoperationen und nötigen Kapselungen den Aufbau des Compilers deutlich. Im folgenden Abschnitt wird daher ein praktikablerer Ansatz geschildert, der auch bisher noch nicht angesprochenen Anforderungen wie der Behandlung partieller Funktionsaufrufe gerecht wird.

3.3 Fortgeschrittene Datentypmodellierung

Das elegantere und letztlich im Prototyp des Compilers verwendete Übersetzungsverfahren für dynamische Datenstrukturen ist nicht unähnlich zum vorher beschriebenen Ansatz. Polymorphe Datentypen und partielle Konstruktor- sowie Funktionsaufrufe können aber so wesentlich einfacher realisiert werden. Dies geschieht durch Definition eines einzigen C-Datentyps, mit dem alle Curry-Datentypen modelliert werden können. Die verwendete C-Struktur besteht aus einer Int-Variable, die als Typidentifikator dient, und einer Union-Struktur, die Variablen für diverse mögliche Daten vereint. Es stehen Variablen für elementare Datentypen (float, int, char) aber auch für Zeiger auf Listen weiterer Datenstrukturen zur Verfügung. Ferner können mittels einer weiteren Struktur partielle Funktionsaufrufe beschrieben werden, worauf später genauer eingegangen wird. In C sieht die Definition des allgemeinen Typs für Datenstrukturen wie folgt aus:

```
struct data
{
   int constr;
   union
   {
     int     idata;/* Variablen für elementare Datentypen */
     char    cdata;
     float   fdata;
     void*   pPartcall;
     data**  args;/* Zeigervariable für Konstruktorargumentfeld */
   };
};
```

Wie leicht zu erkennen ist, kann auf eine Liste weiterer `data`-Strukturen verwiesen werden, wodurch jegliche Curry-Datentypen repräsentiert werden können. Zusätzlich wird vom Compiler ein globales Array generiert, in welchem Stelligkeit, Name des Konstruktors und weitere Informationen festgelegt werden. Der Typidentifikator in der `data`-Struktur dient als Index, mit dem zu jedem Datentyp die zugehörigen Informationen aus dem Konstruktor-Array bestimmt werden können. Am Beispiel des abstrakten Listentyps soll noch einmal die

Funktionsweise genauer aufgezeigt werden. Der im Vorfeld beispielhaft näher erklärte Typ für allgemeine Listen

```
data Liste a   = Element a Liste | LeereListe
```

wird wie folgt in C übersetzt: Zunächst werden die Konstruktoren im globalen Konstruktorfeld eingetragen, in dem auch weiter im Programm verwendete Konstruktoren registriert werden.

```
constrListEl  g_constrList[] =
    {{"LeereListe",0,(void*) &consTest_LeereListeapply},
     {"Element",2,(void*) &consTest_Elementapply} };
     /* in diesem globalen Array würde ebenfalls
          jeder weitere Konstruktor eingetragen */
```

Wobei die Elemente dieses Feldes folgenden Definition haben:

```
struct constrListEl
{
  char* constrname;
  int   arity;
  void* consptr;
};
```

In diesem globalen Feld werden der in Curry verwendete Konstruktorbezeichner, die Stelligkeit des Konstruktors sowie weitere Information, die später näher erläutert werden, gespeichert. Der Bezeichnerstring wird benötigt, um z.B. bei der Darstellung einer Variable mittels des Curry-Befehls `show` ein vollständiges und mit anderen Curry-Systemen (z.B. dem PAKCS) vergleichbares Ergebnis zu erhalten.

Ferner wird wieder für jeden Konstruktor eine C-Funktion erzeugt, welche die Konstruktorparameter geeignet in die oben beschrieben `data`-Struktur einträgt und benötigten Speicher alloziert. Für den Listenkonstruktor hat solch eine Funktion folgendes Aussehen:

```
data *consprelude_Element(data* par0,data* par1)
{
  int  iTypeindex;
  int  iArity;
  int  iStructSize;
  int  iArgsSize;
  data * p_Temp;
  iTypeindex = 1; /* Typindentifikator und
                     Index in Konstruktorfeld */
  iArity = 2;     /* Konstruktorstelligkeit festlegen */
  iStructSize = sizeof(data);
  iArgsSize = sizeof(data*)*iArity;
```

```
            /* Benötige Speichergrösse für Daten berechnen */
  p_Temp = (data *) (prl_alloc(iStructSize));
            /* Speicher für Struktur allozieren */
  p_Temp->constr = iTypeindex;
  p_Temp->args = (data **) (prl_alloc(iArgsSize));
            /* Speicher für Argumentzeiger allozieren*/
  p_Temp->args[0] = par0;   /* Zeiger auf Parameter speichern */
  p_Temp->args[1] = par1;
  return(p_Temp);
}
```

Zusätzlich wird vom Compiler noch eine weitere Funktion generiert, welche zur Behandlung von partiellen Konstruktoraufrufen benötigt wird. Ein Zeiger auf solch eine Apply-Funktion wird ebenfalls im globalen Konstruktorfeld abgelegt. Auf die Behandlung von partiellen Funktionsaufrufen wird in späteren Abschnitten genauer eingegangen. Beim Aufruf einer Konstruktorfunktion muss der Compiler Parameter, die aus elementaren Daten bestehen, geeignet in `data`-Strukturen kapseln und den Zeiger auf die erzeugte Struktur übergeben. Bei allen anderen Datentypen kann einfach der Zeiger auf die beschreibende Struktur übergeben werden, so dass keine Typüberführungen der Zeiger notwendig sind. Für die flexible und einfache Modellierung bezahlt man allerdings den Preis, dass jedes einzelne elementare Datum - zum Beispiel ein Zeichen einer Zeichenkette - in einer `data`-Struktur gekapselt wird, was einen gewissen Overhead mit sich bringt. Gerade bei der Realisierung von Strings ist solch ein Overhead störend, da zudem für jedes Datum einzeln Speicher auf dem Heap alloziert werden muss, der deutlich größer als das Datum ist. Allerdings resultiert dieses Phänomen maßgeblich aus der Tatsache, dass Strings in Curry anders als in C nicht als Felder, sondern als Listen von Zeichen (Chars) realisiert sind, und würde in ähnlicher Form auch bei anderen Übersetzungsstrategien auftreten. Da auf Listen viele Stringoperationen mit weniger Aufwand zu bewerkstelligen sind, ist davon auszugehen, dass insgesamt keine wirklichen Geschwindigkeitseinbußen entstehen.

Durch die Verwendung der `data`-Strukturen für alle Datentypen kann nun, anders als beim naiven Ansatz, leicht eine elegante Vergleichsfunktion implementiert werden. Sie überprüft zunächst, ob die Konstruktor-Ids der Operanden identisch sind, hierbei würde z.B. sofort eine nichtleere Liste von einer leeren unterschieden. Stimmen die Konstruktoren überein, muss nur noch zwischen elementaren Datentypen, die wie gewohnt verglichen werden können, und zusammengesetzten Datentypen unterschieden werden, deren Argumentliste nur elementweise rekursiv überprüft werden muss.

```
/* Curry2C Prelude (==)-operator for data-structures */
int eq_data(data *par0,data *par1)
{
    int typeindex;
    int arity;
```

```
    int c;
    /* Konstruktoren vergleichen */
    if(par0->constr!=par1->constr)
    {
        return(0); /* False*/
    }
    typeindex=par0->constr;

    switch(typeindex)
    {
    /* Elementare Datentypen vergleichen*/
    case consID_INT:
        return((par0->idata)==(par1->idata));
    case consID_FLOAT:
        return((par0->fdata)==(par1->fdata));
    case consID_CHAR:
        return((par0->cdata)==(par1->cdata));
    /* Sonderbehandlung für Partcalls */
    case consID_PARTCALL:
        return(eq_partcall((partcall*)par0->pPartcall,
                           (partcall*)par1->pPartcall));
    /* Rekursiv Argumente vergleichen */
    default:
        arity=g_constrList[typeindex].arity;
        for(c=0;c<arity;c++)
        {
            if(!eq_data(par0->args[c],par1->args[c]))
            {
                return(0);
            }
        }
        return(1);
    }
    return(1);
}
```

Auch die Implementierung der Prelude-Funktion `show` gestaltet sich nun ähnlich einfach wie die der Vergleichsfunktion, wenn auch einige Spezialfälle mehr berücksichtigt werden müssen, die sich aus der Curry-Syntax für Listen und Tupel ergeben.

3.4 Übersetzung von Funktionen

Durch den Verzicht auf die Nachbildung der verzögerten Auswertung von Curry können Curry-Funktionen einfach in C-Funktionen überführt werden. Das Pattern-Matching von Curry wird hierbei durch Switch-Case- Fallunterschei-

dungen nachgebildet. Die Fallunterscheidung für das Pattern-Matching wird bei Funktionsparametern elementaren Typs über die Werte, sonst über den Konstruktorindex der `data`-Struktur geführt. Wie im Falle der Konstruktoren legt der Compiler für alle Funktionen eine globale Datenstruktur an, in dem Bezeichner, Stelligkeit und Funktionszeiger auf die entsprechende Apply-Funktionen vermerkt sind. Für jede Funktion wird zusätzlich eine Apply-Funktion erzeugt, die für partielle Funktionsaufrufe unerlässlich ist.

Die Erzeugung des C-Funktionsrumpfes selbst gestaltet sich recht einfach, da er sich am Aufbau der Curry-Funktionen orientiert. Die Funktionsaufrufe des Curry-Programms werden in entsprechende C-Funktionsaufrufe und Rechenoperationen einfach in die Darstellung mittels äquivalenter C-Operatoren überführt. Stellenweise müssen allerdings geeignete Kapselungen von elementaren Daten in den oben beschriebenen `data`-Strukturen generiert werden, um eine Verwendung in zusammengesetzten Datentypen zu ermöglichen. Analog müssen geeignete Zugriffe in die `data`-Strukturen erstellt werden, welche die Variablen in Curry-Funktionsparametern nachbilden. Folgendes Beispiel soll dies verdeutlichen:

```
headexmpl :: [ a ] -> a
headexmpl ( he : restlist) = he
```

Die Funktion `headexmpl` fordert als Parameter eine nichtleere Liste und gibt den Listenkopf zurück. Diese einstellige Funktion erwartet nach der Übersetzung in C also einen Paramter von Typ `data*`, welcher die Eingabeliste repräsentiert. Um einen Zugriff auf die Variable `he` zu realisieren muß also ein geeigneter Verweis in die `data`-Struktur des Funktionsparameters erzeugt werden, in diesem Fall also ein `(par0->args[0])`. Im ganzen hat die Funktion nach der Übersetzung dieses Aussehen:

```
data *test_headexmpl(data *par0)
{
 switch((par0->constr))
     /* Konstruktor für Pattern-Matching überprüfen */
 {
  case 4:    /* KonstruktorID für nichtleere Liste ? */
    {
     return((par0->args[0])); /* Listenkopf zurückgeben */
    }
  default:
     prelude_failed(); /* Abbruch falls kein Fall anwendbar. */
 }
}
```

Auch die geeignete Bennung der Funktionen muss berücksichtigt werden: In unterschiedlichen Curry-Modulen können gleich lautende Funktionsbezeichner existieren, welche beim Übersetzen in C global und somit im gleichen Namensraum definiert werden würden. Dies würde zu unerwünschten Mehrdeutigkei-

ten führen. Um solche sich aus den Namensbereichen der Curry-Module ergebenden Konflikte zu verhindern, werden die übersetzten Funktionen mit einer Kombination aus Modulnamen und Originalnamen benannt. Der Modulname wird als Präfix des Funktionsnamens verwendet und durch einen Tiefstrich vom Rest des Bezeichners getrennt. So wird zum Beispiel aus dem Bezeichner der Prelude-Funktion `flip` der C-Funktionsname `prelude_flip` generiert. Da es denkbar wäre, dass ein Curry-Bezeichner ebenfalls einen Tiefstrich oder aber in C nicht zulässige Sonderzeichen enthält, werden bei der Übersetzung alle Zeichen im Funktionsnamen, die weder einen Buchstaben noch eine Ziffer darstellen, durch den entsprechenden ASCII-Code in hexadezimaler Schreibweise ersetzt. Zum Beispiel wird der im Curry-Modul `test` befindliche Funktionsbezeichner `function´` in `test_function_27` übersetzt.

3.5 Partielle Funktionsaufrufe

Einen separaten Ansatz bedarf es bei der Übersetzung von partiellen Funktionsaufrufen, also zum Beispiel bei Funktionsaufrufen, für die nicht alle Parameter zur Verfügung gestellt werden. Nativ beherrscht C solche Konstrukte nicht, daher müssen bei der Übersetzung geeignete Strukturen bereitgestellt werden. Die endgültige Ausführung solch eines Aufrufs ist erst möglich, wenn alle benötigen Parameter während des Programmablaufes übergeben wurden. Fehlen Parameter, werden alle bisher gesammelten in einer speziellen Struktur für partielle Funktionsaufrufe gebündelt. In dieser Struktur sind zusätzlich der funktionsspezifische Index zur Adressierung in der globalen Funktionsliste und die Anzahl der noch fehlenden Parameter enthalten. Dieses Konzept orientiert sich im Aufbau an der Behandlung von partiellen Funktionsaufrufen in Flat-Curry. Die benötigte C-Struktur ist wie folgt definiert:

```
struct partcall
{
    int i_Index; /* Funktionsidentifikator */
    int b_isConstrCall;
    int i_MissingPars; /* Anzahl noch fehlender Parameter */
    data**  args;
     /* Zeiger auf vorhandene Parameter in data-Darstellung */
};
```

Dieser `partcall`-Typ wird analog auch für partielle Konstruktoraufrufe verwendet, weshalb das Flag `b_isConstrCall` enthalten ist, welches zur Unterscheidung von Funktions- und Konstruktoraufrufen dient. Der `partcall`-Typ wird wie alle anderen Datentypen auch im oben beschriebenen `data`-Struktur-Typ einbettet.

Die Anwendung partieller Aufrufe erfolgt auch wie in Flat-Curry über die Funktion `apply` der Prelude-Bibliothek, als Input benötigt `apply` eine `partcall`-Struktur, welche alle schon vorhandenen Parameter der anzuwendenden partiellen Funktion enthält, und den hinzuzufügenden Parameter. Ist die Anzahl der benötigten Argumente mit dem neuen Parameter noch nicht erreicht, so wird

dieser zur `partcall`-Struktur hinzugefügt. Andernfalls kann die Funktion ausgewertet werden. Um aus dem Kontext von `apply` die zugehörige Funktion eines Partcalls auswerten zu können, muss aus der globalen Funktionsliste der Zeiger der relevanten Apply-Funktion gelesen werden. Die vom Compiler erzeugten Apply-Funktionen bilden die in der `partcall`-Structur enthaltenen Funktionsparamter auf die zugehörigen Funktion ab. Die Apply-Funktionen stellen also nur ein Hilfskonstrukt dar, um ehemals partielle Funktionsaufrufe, für die nun alle Argumente verfügbar sind, entgültig durch Einsetzen in die relevante C-Funktion auszuwerten. Zum Beispiel wird für die Prelude-Funktion `flip` folgende Apply-Funktion erstellt:

```
data *prelude_flipapply(data **args,data *lastarg)
{
 return(prelude_flip(args[0],args[1],lastarg));
}
```

3.6 Funktionen höherer Ordnung

Die oben beschriebenen Methoden für partielle Funktionsaufrufe liefern die nötige Infrastruktur, um leicht Funktionen höherer Ordnung realisieren zu können. Funktionen höherer Ordnung fordern als Eingabeparameter eine Funktion oder haben eine Funktion als Ergebnis. Ein typisches Beispiel einer Funktion höherer Ordnung ist die Funktion `map` in Curry, welche es ermöglicht, beliebige Operationen auf alle Elemente einer Liste anzuwenden. Hierfür wird dem `map`-Befehl nur die zu bearbeitende Liste und die Funktion zur Bearbeitung der Listenelemente übergeben. `map` führt nun die nötige Iteration über die Listenelemente durch und wendet die übergebene Funktion elementweise an. Es stellt sich die Frage, wie die Übergabe von Funktionen bei der Übersetzung von Curry zu handhaben ist. Zunächst wäre eine einfache Realisierung in C mittels Funktionszeigern denkbar, was allerdings schwierig in die bisher geschilderten Konzepte zu integrieren wäre. Genauer betrachtet stellt die Übergabe von Funktionen aber einen Sonderfall partieller Funktionsaufrufe dar und lässt sich somit leicht realisieren, indem die Abbildungen in `partcall`-Strukturen mit einer leeren Parameterliste eingebettet werden.

3.7 Übersetzungsbeispiel

Am Beispiel des folgenden Curry-Programms sollen die geschilderten Übersetzungsstrategien noch einmal illustriert werden:

```
randomlist=[ 66,30,7,91 ]

qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:l)  = qsort (filter (<x) l) ++ x : qsort (filter (>=x) l)

-- sortieren wiederholen um gut messbare laufzeit
```

```
-- zu erhalten
repeatsort :: Int -> [Int] -> [Int]
repeatsort n l | (n<1)      = l
               | otherwise = (repeatsort (n-1)
                                         (qsort (l++randomlist)))

start =  repeatsort 10 randomlist
```

Durch Übersetzung ergibt sich folgendes C-Programm, wobei die globalen Lookuptables und Apply-Funktionen aus Platzgründen hier nicht berücksichtigt werden:

```
data *test_randomlist()
{
 return(consprelude_List(consprelude_Int(66),
        consprelude_List(consprelude_Int(30),
        consprelude_List(consprelude_Int(7),
        consprelude_List(consprelude_Int(92),
        consprelude_NULL())))));
}

data *test_qsort(data *par0)
{
 switch((par0->constr))
 {
  case 5: /* KonstruktorID fue leere Liste*/
    {
     return(consprelude_NULL());
    }
  case 4: /*KonstruktorID fuer Listen*/
    {
     return(prelude_conc(test_qsort(prelude_filter(
     consprelude_partcall(63,0,1,prelude_addArg(NULL,0,(par0->args[0]))),
     (par0->args[1]))),
     consprelude_List((par0->args[0]),
     test_qsort(prelude_filter(consprelude_partcall(64,0,1,
     prelude_addArg(NULL,0,(par0->args[0]))),(par0->args[1])))));
    }
  default:
     prelude_failed();
 }
}


data *test_repeatsort(int par0,data *par1)
{
 if((par0<1))
```

```
   {
    return(par1);
   }
 else
   {
    return(test_repeatsort((par0-1),
           test_qsort(prelude_conc(par1,test_randomlist()))));
   }
}

/* Die folgenden zwei Funktionen wurden als Ersatz für die anonymen
   Vergleichfunktionen generiert. Sie werden hier nicht direkt aufgerufen,
   sondern nur über ihre FunktionsID (63,64) in den entsprechenden Partcall-
   Strukturen in qsort verwendet!
*/

int test_test_5flambda0_5f0(int par0,int par1)
{
 return((par1<par0));
}

int test_test_5flambda1_5f0(int par0,int par1)
{
 return((par1>=par0));
}

/* nullstellige Startfunktion  */
data *test_start()
{
 return(test_repeatsort(10,test_randomlist()));
}

int main(int argc,char *argv[])
{
 printf(__curry2cstring(prelude_show(test_start2())));
 return(0);
}
```

Das Code-Beispiel macht deutlich, dass der generierte Code im Vergleich z.B. zum Münster-Curry- Compiler recht gut lesbar ist. Es wäre durchaus möglich, solchen Code mit vorhandenem C-Code zu kombinieren. Auch Optimierungen durch den Entwickler scheinen angesichts der Lesbarkeit nicht abwegig.

3.8 Einschränkungen

Der in dieser Arbeit beschriebene experimentelle Ansatz zur direkten Übersetzung von Curry in strikte Programmiersprachen bringt, wie schon angedeutet, einige Einschränkungen mit sich. Zum einen werden wichtige Fähigkeiten von Curry, z.B. Constraint-Solving, nicht unterstützt. Diese Reduktion des Funktionsumfanges, gerade auch der Verzicht auf Lazy-Auswertungsstrategien, war Teil des Experimentes und von vornherein beabsichtigt. Aus der direkten und strikten Implementierung von Funktionsaufrufen resultieren allerdings weitere Einschränkungen. Realisiert man in Curry reaktive Programme oder auch iterative Algorithmen durch (End-)Rekursion, so verwenden die übersetzten C-Programme ebenfalls Endrekursion. Bei sehr großen Datenmengen oder eben bei der nichterminierenden Endrekursion reaktiver Programme können so Stack-Überläufe entstehen. Dieses Problem resultiert aus der Art, in der C Funktionsaufrufe realisiert, und beschäftigt Entwickler von Compilern zur Übersetzung von funktionalen Sprachen in C immer wieder[9][2]. Es wären also generell andere Aufrufstrategien nötig, oder aber es müsste versucht werden, dem Problem mit einer geeigneten Optimierung zu begegnen, die solche rekursiven Aufrufe z.B. durch Sprünge oder Schleifenkonstrukte ersetzt. Derartige interessante Weiterentwicklungen wären aber über die Zielsetzungen dieser Studienarbeit hinausgegangen. Durch die relativ gute Lesbarkeit des erzeugten C-Codes können allerdings relativ problemlos benötigte Änderungen an den Programmen vorgenommen werden, und z.B. durch manuelles Einfügen von Schleifen reaktive Programme ermöglicht werden.

3.9 Optimierungen

Bei der Generierung des C-Codes ergeben sich nahe liegende Möglichkeiten zur Optimierung. In Flat-Curry wird eine Fallunterscheidung mittels `if_then_else` als dreistellige Funktion des Prelude-Moduls dargestellt. Bei einer direkten Übersetzung nach obigem Schema, müsste also die C-Prelude auch solche eine dreistellige Funktion beinhalten. Ein deratiger Ansatz scheint allerdings wenig sinnvoll, daher wird `if_then_else` vom Compiler gesondert behandelt und in das C-Äquivalent `if_else` überführt. Ähnliches gilt auch für den Curry-Typ `Bool`, welcher streng genommen in der oben beschriebenen Form in einen C-Typ übersetzt werden müsste. Aufgrund der elementaren Bedeutung boolscher Variablen, werden diese wie üblich in Variablen vom C-Typ `int` überführt, da C selbst keinen eigenen Typ für boolsche Werte hat. Diese Überführung kann bedenkenlos ausgeführt werden, weil der Compiler für die Übersetzung auf korrekt getypte Flat-Curry-Dateien zurückgreift, und somit keine Gefahr durch falsche Typüberführungen droht.

Eine weitere Optimierung wäre durch Handoptimierung des C-Codes der Prelude-Funktionen denkbar. Der größte Teil der Prelude-Funktionen wurde falls möglich mittels des Curry2C-Compiler erstellt und nur stellenweise manuell angepasst. Dadurch sind z.B. iterative Listenoperationen auch in der C-Variante durch Endrekursion implementiert. Solche Rekursionen könnten leicht

von Hand in Schleifenkonstrukte überführt werden. Die Reduzierung von rekursiven Funktionsaufrufen sollten sich bei größeren Datenmengen durch einen Geschwindigkeitszuwachs auszeichnen. Ferner könnte so ein Ansatz zumindest bei Prelude-Funktion die Gefahr möglicher Stack-Overflows reduzieren.

4 Curry2C Implementierung

Die beschrieben Übersetzungstechniken wurden in einem Compiler-Prototyp implementiert. Der entstandene Curry2C-Compiler ist selbst in Curry realisiert. Die zu compilierenden Curry-Programme werden zunächst mittels des Flat-Curry-Moduls des PAKCS in Flat-Curry überführt. Die Repräsentation in Flat-Curry ermöglicht eine leichtere Behandlung der Programme, da diese nach der Überführung als Curry-Datentypen vorliegen. Ferner werden bei der Umwandlung von Curry in Flat-Curry syntaktische Vereinfachungen vorgenommen, von denen der Curry2C-Compiler profitiert. Basierend auf der Flat-Curry-Darstellung werden dann die Überführung der definierten Datentypen und Funktionen vorgenommen. Bei der Entwicklung des Compilers wurde zunächst die Idee verfolgt, durch Repräsentation der vom Compiler erzeugten Programmteile und Deklaration in einer weiteren Zwischensprache, ein späteres Übersetzen in beliebige imperative Sprachen zu ermöglichen. Aus diesem Grund wird das Flat-Curry-Programm zunächst in eine solche weitere imperative Zwischensprache überführt, aus der dann abschließend der C-Code generiert wird. Es zeigte sich allerdings, dass es hilfreich ist, sich bei der Definition der Zwischensprache an C zu orientieren, um eine möglichst einfache direkte Überführung in C erreichen zu können. Diese Nähe zu C könnte das Generieren von Code anderer Sprachen, die z.B. keine Zeigerarithmetik unterstützen, eventuell erschweren.

Für die benötige Speicherbereinigung der erstellten Programme verwendet der Curry2C-Compiler den verbreiteten Garbage-Collector (GC) von Boehm, Demers und Weiser [12]. Hierfür wird statt des üblichen Allozierungsbefehls `malloc` die GC-eigene Funktion verwendet, eine weitere Integration des Garbage-Collectors in den erzeugten Code ist nicht nötig.

Um ein ausführbares C-Programm erhalten zu können, muss die Funktion `main()` definiert sein, welche den Einsprungspunkt des Betriebssystem in das C-Programm darstellt. In ihr wird festgelegt, welche der übersetzen Curry-Funktionen ausgewertet werden soll. Dazu wird dem Curry2C-System vor dem Kompilieren eine nullstellige Curry-Funktion genannt werden, welche von `main()` aus aufgerufen werden soll. Das Ergebnis dieser Funktion wird über die Standardausgabe dargestellt, wobei die Formatierung des Ergebnisstrings mittels des `Show`-Befehls der Prelude erzeugt wird, um eine mit anderen Curry-System vergleichbare Darstellung zu erzielen.

Die aktuelle Implementierung des Curry2C-Compilers besitzt noch einige Einschränkungen. So fehlen z.B. die Möglichkeiten zur Benutzung von Currys IO-Operationen, und eine Einbindung von extern definierten Funktionen ist nicht möglich. Diese würden sich relativ leicht realisieren lassen, waren aber für die Durchführung der angestrebten Experimente im Rahmen dieser Arbeit nicht notwendig. Diese Arbeit stellte eine Machbarkeitsstudie dar und hatte nicht die vollständige Implementierung der Basismodule Currys als Zielsetzung. Auch muss darauf hingewiesen werden, dass möglicherweise einige reine C-Compiler den von Curry2C erzeugten Code als fehlerhaft ansehen könnten, weil Typdeklarationen der Einfachheit halber der Notation von C++ folgen. Dieser Mängel kann aber bei Bedarf leicht abgestellt werden, allerdings hatten die in dieser

Arbeit getesteten Compiler mit der verwendeten Syntax keine Schwierigkeiten .

Als Ergebnis legt der Compiler zwei Dateien an. Zum einen wird ein C-File erstellt, welches die direkten Übersetzungen der Funktionen,Konstruktoren und Datentypen enthält, zum anderen wird eine Datei generiert, die die Vorabdeklarationen, die globalen Listen für Konstruktoren und Funktionen und die angesprochenen Apply-Funktionen enthält. Die Notwendigkeit für die Aufteilung des produzierten Codes auf zwei Dateien ergibt sich aus diversen Abhängigkeiten. Die Funktionen der Prelude, die dem Compiler als C-File beiliegen, benötigen Zugriff auf die globalen Funktions- und Konstruktorfelder. Also müssen diese Definitionen von der Prelude importiert werden, welche wiederum vom Hauptprogramm benötigt wird. Natürlich wäre es denkbar die Felder einfach über zur Übersetzungszeit leere Zeiger zu repräsentieren, und diese dann zur Laufzeit geeignet zu setzen, allerdings würde dieser Weg das Hauptprogramm verkomplizieren. Ferner scheint es auch aus Gründen der Übersichtlichkeit sinnvoll, die angesprochene Trennung beizubehalten und so die relativ gute Lesbarkeit des von Curry2C erzeugten Codes nicht weiter zu beeinträchtigen.

4.1 Anwendungsbeispiel des Curry2C-Compilers

Nun soll mittels eines einfachen Beispiels die Benutzung des Curry2C-Systems erklärt werden. Das PAKCS, ein geeigneter C-Compiler, z.B. der GCC und die jeweilige systemspezifische Variante des Boehm-Garbage-Collectors werden in diesem Beispiel als dem Nutzer vertraut und installiert vorausgesetzt. Es soll nun das Beispielprogramm `test.curry` aus dem vorherigen Kapitel zunächst in ein C-Programm übersetzt und dieses dann in eine ausführbare Datei überführt werden. Als weitere Vorbereitung ist es hilfreich, alle Dateien des Curry2C-Paketes und das zu übersetzende Programm in einem Verzeichnis abzulegen. Die folgenden Schritte können dann in diesem Verzeichnis ausgeführt werden:

1. Das PAKCS starten.
2. Den Curry2C-Compiler durch Eingabe von

   ```
   :l Fcy2CFlat
   ```

   in das PAKCS laden. Der Curry2C-Compiler steht nun zur Verfügung.
3. Das Übersetzen des Programms erfolgt mittels des Befehls `convert2C`. Die Funktion `convert2C` fordert zwei Parameter: den Namen des zu übersetzenden Curry-Programms und den Bezeichner der beim Start des Programms aufzurufenden Curry-Funktion. Wie schon erklärt, muss diese Funktion eine Stelligkeit von Null aufweisen. Im beschriebenen Beispielprogramm ist dies die Funktion `start`. Ihr Bezeichner ist als Paar aus Modulname und Funktionsname anzugeben. Der Aufruf von `convert2C` hat dann folgende Form:

   ```
   convert2C "test" ("test","start")
   ```

Der Curry2C-Compiler erzeugt jetzt - falls keine Fehler auftreten - die Dateien `test.cc` und `cflatglobals.cc` im aktuellen Verzeichnis.

4. Das PACKS mit `:q` verlassen und die erstellten C-Dateien übersetzen. Wird zum Beispiel als Compiler der GCC verwendet, kann die Übersetzung in etwa wie folgt gestartet werden:

   ```
   gcc test.cc -otest.exe -lgc
   ```

   Es wird nun die Programmdatei `test.exe` erstellt.

5 Benchmarks

Abschließend sollen die vom Prototyp des Curry2C-Compilers erzeugten Programme mit denen anderer Curry-Compiler verglichen werden. Als Vergleichscompiler wurden das Portland Aachen Kiel Curry System (PAKCS) in der Version 1.6.1-4 [10] und der Münster Curry Compiler (MCC)[11] in der Version 0.9.8 betrachtet. Die Übersetzung der Curry2C-Programme wurde mit dem GCC 2.95.3 unter Verwendung von Optimierungen der Stufe 2 (Parameter -O2) durchgeführt. Getestet wurden das Umkehren einer tausendelementigen Integer-Liste durch den Naive-Reverse-Algorithmus, Sortieren einer Integer-Liste mit 120 Elementen mittels des Insertion-Sort-Algorithmus und Berechnung der 30. Fibonaccizahl durch naive Rekursion. Um aussagekräftige Laufzeiten zu erhalten, wurden das Listenumkehren hundertmal und das Sortieren hunderttausendmal wiederholt. Die Laufzeiten wurden mittels des Befehls `time` unter SunOS 5.9 bestimmt.

Es zeigt sich, dass der vom Curry2C-Compiler erzeugte Code in allen drei Tests dem PAKC-System in Geschwindigkeit deutlich überlegen ist. Differenzierter sieht der Vergleich mit dem Münster-Curry-Compiler aus: In den Benchmarks, die das häufige Allozieren von Speicher benötigen (Reverse und Insertion Sort), liegt der MCC vorn. Beim rekursiven Berechnen der Fibonaccifunktion scheint der Curry2C-Compiler von der direkten Implementierung der Funktionsaufrufe zu profitieren, und liegt im Vergleich vor allen anderen Compilern. Offenbar wird der Curry2C-Code im Gegensatz zum MCC durch die Allozierungsoperationen gebremst. Beim Vergleich der Dateigröße schneidet der Curry2C-Compiler selbst ohne explizite Größenoptimierung beim Erzeugen der Executables gut ab. Alle erzeugten Testprogramme sind deutlich kleiner als die ausführbaren Dateien der anderen Compiler. Es darf dabei allerdings nicht verschwiegen werden, dass zum Ausführen der Curry2C-Programme die Bibliothek des Garbage-Collectors im System installiert sein muss, deren Dateigröße evtl. auch berücksichtigt werden müsste. Allerdings ist die Größe der Bibliothek fix, und verliert mit zunehmender Zahl von darauf zugreifenden Programmen an Bedeutung.

	MCC	PAKCS	Curry2C
Dateigröße (kb)	653	1191	45
Laufzeit (sek)	26,39	64,24	45,25

Tabelle 1: Benchmarkergebnisse und Dateigrößen für Naive Reverse

	MCC	PAKCS	Curry2C
Dateigröße (kb)	687	1188	40
Laufzeit (sek)	21,72	336,41	24,91

Tabelle 2: Benchmarkergebnisse und Dateigrößen für Insertion Sort

	MCC	PAKCS	Curry2C
Dateigröße (kb)	664	1189	36
Laufzeit (sek)	3,018	40,8	0,054

Tabelle 3: Benchmarkergebnisse und Dateigrößen für Fib(30)

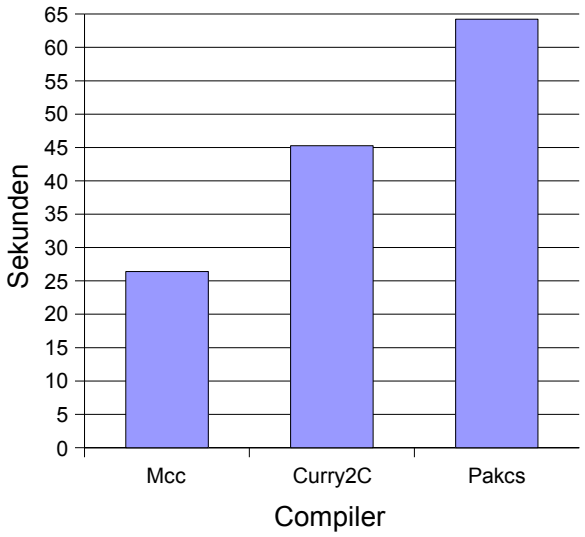

Abbildung 1: Benchmarkergebnisse für Naive-Reverse

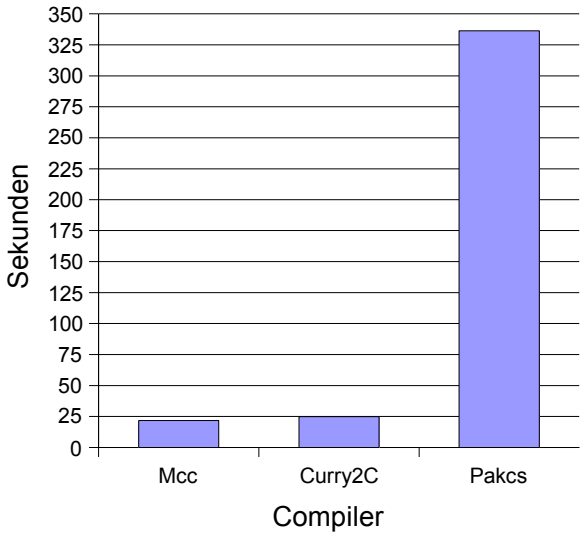

Abbildung 2: Benchmarkergebnisse für Insertion-Sort

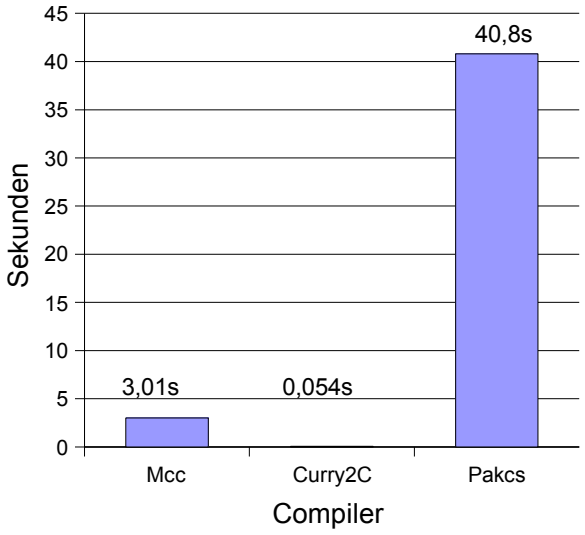

Abbildung 3: Benchmarkergebnisse für Fib(30)

```
-- naive reverse
append []     ys = ys
append (x:xs) ys = x : append xs ys

rev []     = []
rev (x:xs) = append (rev xs) [x]

buildlist :: Int -> [Int]
buildlist n | (n<1)     = []
            | otherwise = (n:(buildlist (n-1)))

repeatReverse :: Int -> [Int] -> [Int]
repeatReverse n l | (n<1) = l
                  | otherwise = (repeatReverse (n-1) (rev l))

main = repeatReverse 100 (buildlist 1000)
```

Abbildung 4: Curry-Code des Naive-Reverse-Benchmarks

```
randomlist=[ 41,67,34,0,69,24,78,58,62,64,5,45,81,27,61,91,95,42,27,36,
             91,4,2,53,92,82,21,16,18,95,47,26,71,38,69,12,67,99,35,94,
             3,11,22,33,73,64,41,11,53,68,47,44,62,57,37,59,23,41,29,78,
             16,35,90,42,88,6,40,42,64,48,46,5,90,29,70,50,6,1,93,48,
             29,23,84,54,56,40,66,76,31,8,44,39,26,23,37,38,18,82,29,41,
             33,15,39,58,4,30,77,6,73,86,21,45,24,72,70,29,77,73,97,12,
             86,90,61,36,55,67,55,74,31,52,50,50,41,24,66,30,7,91,7,37]

sort [] = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) | x <= y = x:y:ys
                | otherwise = y : insert x ys

repeatSort :: Int -> [Int] -> [Int]
repeatSort n l | (n<1) = l
               | otherwise = (repeatSort (n-1) (sort l))

start = repeatSort 100000 randomlist
```

Abbildung 5: Curry-Code des Insertion-Sort-Benchmarks

```
fibonacci :: Int -> Int
fibonacci n | n<1 = 0
            | n==1 = 1
            | n==2 = 1
            | n>2 = (fibonacci (n-2)) + (fibonacci (n-1))

start = fibonacci 30
```

Abbildung 6: Curry-Code des Fibonacci-Benchmarks

6 Fazit

Es wurde ein Weg aufgezeigt, mit dem geeignete in Curry entwickelte Programme in C übersetzt werden können, wobei diese Übersetzung möglichst direkt und ohne Modellierung einer abstrakten Maschine realisiert wird. Die beschriebenen Techniken lassen sich in groben Zügen auch für die Übersetzung in andere strikte imperative Sprachen anwenden. Das Ignorieren eines Teils des Funktionsumfangs von Curry ermöglicht die Erzeugung von sehr schlanken Executables, und kann auch Performanzvorteile mit sich bringen. Allerdings zeigen Vergleiche mit mächtigeren Compilern wie dem MCC, dass die vorgenommen Einschränkungen nicht zwangsläufig in Geschwindigkeitsvorteilen resultieren. Trotzdem stellt der vorgestellte Ansatz einen vielversprechenden Weg zur Implementierung von Programmen mit Curry für Systeme mit eingeschränkten Ressourcen dar. Bei der Entwicklung von z.B. Webanwendungen oder Software für eingebettete Systeme kann dieses Zusammenspiel von geringer Dateigröße und guter Performanz der von Curry2C erzeugten Programme hilfreich sein.

Literatur

[1] M.Hanus. Curry - An Integrated Functional Logic Language. `http://www.informatik.uni-kiel.de/~curry`

[2] M.Hanus, K. Höppner, F. Huch. Towards Translating Embedded Curry to C. Electr. Notes Theor. Comput. Sci. 86(3), 2003

[3] F. Henderson, T. Conway, Z. Somogyi. Compiling logic programs to C using GNU C as portable assembler. Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming, 1995, Portland, Oregon. pp. 1-15

[4] H. Boehm, M. Weiser. Garbage Collection in an Uncooperative Environment, Software Practice and Experience 18, 1988, pp.807-820

[5] W. Lux, H. Kuchen. An Efficient Abstract Machine for Curry. Informatik '99 — Informatik überwindet Grenzen, 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 5.–9. Oktober 1999

[6] M. Hanus, F. Huch. An Open System to Support Web-based Learning. Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)

[7] S. P. Jones, N. Ramsey, F. Reig. C–: a Portable Assembly Language that Supports Garbage Collection. International Conference on Principles and Practice of Declarative Programming. 1999

[8] D. Tarditi, A. Acharya, P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990

[9] J. Bartlett. Scheme->C: A portable Scheme-to-C compiler. Tech. Rept., DEC West. Res. Lab., 1989.

[10] Portland Aachen Kiel Curry System - `http://www.informatik.uni-kiel.de/~pakcs/`

[11] Münster Curry Compiler - `http://danae.uni-muenster.de/~lux/curry/`

[12] Boehm-Demers-Weiser-Garbage-Collector - `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`