

C++ Parser sind universell

Seminar „Perlen der theoretischen Informatik“ der AG Meyer auf der Heide im WS2004/2005

Matthias Hilbig 6079733
Betreuer: Martin Ziegler

17. Januar 2005

1 Einführung

Die Syntax von C++ ist kompliziert . . . und mächtig. Wer hätte schon vermutet, dass man schon beim Übersetzen von C++ Programmen beliebige Berechnungen anstellen kann? Die Sprache ist schließlich ohne das Prüfen der Typen kontextfrei¹. Der Erfinder von C++ Bjarne Stroustrup tat das auf jeden Fall nicht, als Erwin Unruh ihn 1994 auf einem C++ Standardisierungstreffen auf diese Idee aufmerksam machte. Am nächsten Tag konnte Erwin Unruh dem verblüfften Bjarne Stroustrup ein Programm zeigen, das Primzahlen während der Übersetzung berechnete. Obwohl Erwin Unruh einige Wochen später den Beweis entwickelte, dass der Template-Mechanismus Turing-vollständig ist, veröffentlichte er diesen nicht. Den Beweis veröffentlichten unabhängig voneinander Todd Veldhuizen [Vel03] und Martin Böhme und Bodo Manthey [BM03] im Jahre 2003.

Diese Ausarbeitung behandelt hauptsächlich den Beweis von Martin Böhme und Bodo Manthey. Dazu werden zunächst die Grundlagen erklärt. Einem Abstecher in die Syntax von C++ Templates und Typdefinitionen folgt ein etwas theoretischerer Abschnitt über partiell rekursive Funktionen. Im letzten Abschnitt schließlich wird erklärt, wie man diese beiden Sachen kombinieren muss, um Endlosschleifen mit dem Übersetzer zu berechnen.

2 C++ Templates

Templates² dienen bei C++ dazu Klassen für verschiedene Typen auszulegen. Anstatt zum Beispiel eine Liste einmal für `int` und einmal für `String`

zu schreiben, kann man dieses einfach mit Templates verallgemeinern und muss im Endeffekt die Liste nur einmal schreiben. Welche Typen später in dieser Liste sind, ist dann egal. Genug der Vorrede: ein Beispiel macht die Syntax und Benutzung von C++ Templates am besten deutlich. Klassen werden in C++ normalerweise mit dem Schlüsselwort `class` deklariert, eine Deklaration mit `struct` ist jedoch gleichwertig bis auf den Unterschied, dass bei einer `struct` alle Objektvariablen standardmäßig `public` sind. Dies macht die Beispiele für unseren Zweck einfacher lesbar.

```
template<class T> struct example {  
    T data;  
    example() {}  
  
    void setData(T d) {data = d;}  
    T getData() {return data;}  
};
```

Der Klasse `example` aus dem Beispiel wurde die Template Deklaration `template<class T>` vorangestellt. Diese Deklaration bedeutet, dass jedes Vorkommen von `T` in der Klassendeklaration später durch den Typ ersetzt werden soll, mit dem die Klasse definiert wird. Wenn man die Klasse `example` später mit dem Typ `int` benutzen möchte schreibt man:

```
int zahl = 1;  
example<int> data;  
data.setData(zahl);
```

Der C++ Übersetzer wird die Klasse `example<int>` so kompilieren, als hätte man die Klasse direkt für den Typ `int` geschrieben, d.h. jedes Vorkommen von `T` in `example` wird durch `int` ersetzt. Die folgende Klasse ist daher äquivalent zu `example<int>`:

```
struct example_int {  
    int data;
```

¹Kontextfreie Sprachen lassen sich mit dem CYK Algorithmus in Zeit $O(n^3)$ parsen [You67]; eine kontextfreie Grammatik für C++ findet sich zum Beispiel in [Str97] oder [Int03].

²Die deutsche Übersetzung „Schablone“ ist in diesem Fall sogar mal nicht irreführend, dennoch wird im folgenden weiterhin das Wort `Template` benutzt, um den Wiedererkennungswert zu erhöhen.

```

example() {}

void setData(int d) {data = d;}
int getData() {return data;}
};

```

Das Ersetzen von Typen in Templates reicht allein aber noch nicht zum Ausführen von beliebigen Funktionen. Wir benötigen dafür noch die Technik der Spezialisierung.

Unter der Spezialisierung eines Templates versteht man die Möglichkeit, eine Klasse für einzelne Typen zu spezialisieren. Man möchte zum Beispiel, dass die Vergleichsfunktion einer Klasse für den Typ `int` anders funktioniert als für den Typ `String`.

Wir werden das obige Beispiel nun für den Typ `String` spezialisieren. Zuerst müssen wir das Template dem Übersetzer mit einer *forward* Deklaration bekanntmachen:

```
template<class T> struct example;
```

Jetzt folgt die Deklaration für den Spezialfall `String`

```
template<> struct example<String> {
    //String spezifische Klasse
};
```

Und schließlich müssen wir noch den allgemeinen Fall für alle anderen Typen angeben

```
template<class T> struct example {
    //Klasse die für alle anderen
    //Typen gilt
};
```

Der Übersetzer entscheidet nun anhand des Typs, den das Template übergeben bekommt, welche der beiden Klassendeklarationen angewandt wird.

Dies ist noch nicht alles, was es über C++ Templates zu sagen gäbe. Templates sind insgesamt recht komplex in C++. Zum Verständnis der in dieser Arbeit benutzten Templates reicht aber das bisher Gesagte.

2.1 C++ Typdefinitionen

Um lange Typnamen zu vermeiden kann man unter C++ „neue“ Typen definieren. Dabei kann man nur schon bestehenden Typen neue Namen zuweisen. Die Syntax sieht dann folgendermaßen aus:

```
typedef int Zahl;
```

In diesem Beispiel wurde ein neuer Datentyp `Zahl` angelegt, der nichts anderes als ein `int` ist. Interessant werden Typdefinitionen im Zusammenspiel mit

Templates. Bei Templates ist es häufig so, dass sehr lange Namen entstehen, mit einem `typedef` kann man diese abkürzen.

```
typedef example<String> exString;
```

Ob die abgekürzten Namen dann besser verständlich als die Originalnamen sind, ist eine andere Sache. Noch eine Stufe komplizierter wird es, wenn man Typdefinitionen innerhalb von Templates benutzt und dabei den Typ verwendet, der dem Template übergeben wurde.

```
template<class T> struct suc {
    typedef T pre;
};
```

In der Klasse `suc` wird der Name `pre`, der an das Template übergebenen Klasse `T` zugewiesen. Wenn man auf komplizierte Weise eine `int` Variable erzeugen will, sollte man also

```
suc<int>::pre i;
```

verwenden. Man sieht, dass man die Klasse `suc` nicht instantiiieren muss, um auf die Typdefinition zuzugreifen. Es reicht den Namen der Template Klasse hinzuschreiben, um dann mit `::pre` die enthaltene Typdefinition zu benutzen. Wir wollen die enthaltene Typdefinition jetzt nicht nur für Variablen sondern auch in weiteren Typdefinitionen benutzen. Dazu benötigt man das Schlüsselwort `typename`, das dem Übersetzer sagt, dass es sich beim nächsten Begriff um einen Typ handelt.

```
typedef typename suc<int>::pre Zahl;
```

Dieses Beispiel ist äquivalent zum einführen des Beispiel `typedef int Zahl;`, da `::pre` von `suc<int>` wieder den Typ `int` ergibt.

3 Partiiell rekursive Funktionen

Ziel dieser Ausarbeitung ist es zu erläutern wie bereits beim Übersetzen von C++ Programmen beliebige partielle Funktionen berechnet werden können. Was aber sind partiell rekursive Funktionen und lässt sich mit denen all das berechnen, was man auch mit Turingmaschinen berechnen kann? Sind partiell rekursive Funktionen also Turing-vollständig? Wer schon einmal in LISP oder ähnlichen funktionalen Programmiersprachen programmiert hat, wird große Ähnlichkeit zu partiell rekursiven Funktionen feststellen. Die zugrunde liegende Struktur von partiell rekursiven Funktionen basiert nicht auf der Grundlage von Maschinen, die Register oder Zellen zur

$$f(x_1, \dots, x_n, k) = \underbrace{h(x_1, \dots, x_n, h(x_1, \dots, x_n, \dots h(x_1, \dots, x_n, g(x_1, \dots, x_n))))}_{k\text{-mal}}$$

Abbildung 1: Auflösung der primitiven Rekursion

Verfügung haben, sondern auf dem etwas abstrakteren Modell, dass alles durch Funktionen beschrieben werden kann. Es gibt also keine Instruktionen, die nacheinander abgearbeitet werden, sondern nur Funktionen mit beliebig vielen Argumenten, die wiederum andere Funktionen aufrufen können und die am Ende einen Wert zurückliefern. Zunächst benötigen wir die Basisfunktionen, die die grundlegenden Operationen angeben, die möglich sind.

Definition 3.1. Zur Klasse der Basisfunktionen gehören:

1. die Nullfunktion: $x \mapsto 0$
2. die Nachfolgerfunktion: $x \mapsto x + 1$ für alle $x \in \mathbb{N}$
3. die Projektionsfunktion: $(x_1, \dots, x_n) \mapsto x_j$ für alle $x_1, \dots, x_n \in \mathbb{N}$

Mit diesen Funktionen alleine kann man noch nicht viel anfangen. Es lassen sich lediglich beliebige Konstanten erzeugen, indem man die Nachfolgerfunktion entsprechend oft auf die Nullfunktion anwendet. Die Konstante 6 lässt sich daher folgendermaßen erzeugen:

$$((((0) + 1) + 1) + 1) + 1) + 1) + 1)$$

Mehr Schwung in die Sache kommt mit den folgenden Operatoren.

Definition 3.2. Seien die beiden primitiv rekursiven Funktionen g und h mit

$$g : \mathbb{N}^n \rightarrow \mathbb{N}$$

und

$$h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$$

gegeben. Dann ist auch die Funktion f mit

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y + 1) &= h(x_1, \dots, x_n, f(x_1, \dots, x_n, y)) \end{aligned}$$

primitiv rekursiv.

Wie funktioniert diese Funktion f ? Klar ist, dass f so etwas wie eine Schleife darstellt. Tatsächlich ist es so, dass die Funktion h insgesamt $(y + 1)$ -mal rekursiv aufgerufen wird, bevor g ausgeführt wird. Es ergibt sich dabei die in Abbildung 1 gezeigte Form.

Außerdem benötigen wir noch die Komposition:

Definition 3.3. Seien g_1, \dots, g_m primitiv rekursiv, jeweils n -stellig, und sei h eine m -stellige primitiv rekursive Funktion. Dann ist auch die Funktion f mit

$$f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

primitiv rekursiv.

Mit dieser Operation ist es nun möglich Ergebnisse von Funktionen zu kombinieren. Die Definition sagt im Prinzip nichts anderes aus, als dass man den Wert, den eine Funktion zurückliefert, als Argument einer anderen Funktion benutzen kann.

Mit diesen beiden Operationen kann man schon einige nützliche Funktionen definieren. Um ein Gefühl für diese Operatoren zu bekommen daher nun einige Beispiele.

Zunächst die Vorgängerfunktion $P(x) = x - 1$: Wir nehmen die Nullfunktion als Funktion g und $h(x) = x$ für die Funktion h . $P(x)$ ergibt sich nun durch primitive Rekursion aus g und h .

$$\begin{aligned} P(0) &= g(0) = 0 \\ P(x + 1) &= h(x, P(x)) = x \end{aligned}$$

Für den Grenzfall $P(0)$ ergibt sich als Vorgänger also ebenfalls 0 und für alle anderen Werte x erhält man das gewünschte $P(x) = x - 1$. Bemerkung: Wenn man diese Funktion nachvollzieht bemerkt man, dass die Rekursion eigentlich gar nicht genutzt wird, da alle Zwischenergebnisse einfach verworfen werden.

Beim folgenden Beispiel werden auch die Zwischenergebnisse benötigt: Wir wollen zwei Zahlen addieren. Dies geschieht wiederum durch einmalige Anwendung der primitiven Rekursion auf zwei Funktionen g und h . Zusätzlich benötigen wir dieses Mal auch die Komposition von Funktionen, da die Funktion h aus einer Komposition von Projektion und Inkrementation besteht:

$$h(x_1, x_2, x_3) = x_3 + 1$$

Die Funktion g gibt im Falle der Addition den ersten Parameter zurück:

$$g(x_1, x_2) = x_1$$

Insgesamt sieht die Addition also folgendermaßen aus:

$$\begin{aligned} A(x_1, 0) &= x_1 \\ A(x_1, y + 1) &= A(x_1, y) + 1 \end{aligned}$$

Die zweite Bedingung wird $(y + 1)$ -mal aufgerufen, die erste Bedingung erst ganz am Ende der Rekursion. Es ergibt sich:

$$A(x_1, x_2) = (x_1) + \underbrace{1 + 1 \cdots + 1}_{x_2\text{-mal}} = x_1 + x_2$$

Auf ähnliche Weise lässt sich eine ganze Klasse von Funktionen definieren. Mit den Basisfunktionen und diesen beiden Operatoren erhalten wir nun die Klasse der primitiv rekursiven Funktionen.

Definition 3.4. Die Klasse der primitiv rekursiven Funktionen ist die kleinste Klasse von Funktionen, die alle Basisfunktionen enthält und die unter den Operationen Komposition und primitiver Rekursion abgeschlossen ist.

Viele Funktionen lassen sich mit der Klasse der primitiv rekursiven Funktionen berechnen, aber immer noch nicht all jene, die sich auch mit einer Turingmaschine berechnen lassen. Es ist zum Beispiel nicht möglich mit primitiver Rekursion eine Endlosschleife zu programmieren. Dazu fehlt eine Art Schleife, bei der man mithilfe einer Funktion selber bestimmen kann, wann die Schleife beendet werden soll. Die Funktion zur Steuerung der Schleife bezeichnet man auch als Prädikat, da die Funktion nur die beiden Werte 0 und 1 für falsch und wahr zurückliefert.

Definition 3.5. Gegeben ein (partielles) Prädikat p mit $p : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$. Dann ergibt sich f durch μ -Rekursion aus p , wenn

$$f(x_1, \dots, x_n) = \mu y [p(x_1, \dots, x_n, y) = 0]$$

gilt, d.h. f liefert das kleinste y zurück für das $p(x_1, \dots, x_n, y) = 0$ ist, wobei aber $p(x_1, \dots, x_n, i)$ für alle $i < y$ definiert sein muss. Wenn so ein y nicht existiert, ist $f(x_1, \dots, x_n)$ undefiniert.

μ -Rekursion stellt also so etwas wie eine WHILE-Schleife dar. Um das minimale y zu finden, müssen die Berechnungen in p auch für alle Zahlen gemacht werden, die kleiner als y sind. Es gibt jedoch keine Möglichkeit, an die Ergebnisse der Berechnungen zu kommen, die p durchführt. Um eine richtige WHILE-Schleife zu bekommen, wendet man μ -Rekursion zusammen mit primitiver Rekursion an. Im ersten Schritt ermittelt man mithilfe der Schleifenbedingung und der μ -Rekursion, bis zu welchem y die Schleife laufen würde (Wenn die Schleifenbedingung von den Berechnungen in der Schleife abhängt, muss man die natürlich auch ausführen). Der zweite Schritt benutzt dieses y als Eingabe für eine primitive Rekursion, die die Schleifenbedingung dann y -mal ausführt und das Ergebnis zurückliefert. Man

sieht, dass man die Berechnungen in der Schleife möglicherweise doppelt durchführt. Die Laufzeit ist bei unseren Betrachtungen aber nebensächlich, es geht alleine darum, was berechnet werden kann und was nicht. Mit der μ -Rekursion können wir nun die Klasse der partiell rekursiven Funktionen definieren.

Definition 3.6. Die Klasse der partiell rekursiven Funktionen ist die kleinste Klasse von Funktionen, die alle Basisfunktionen enthält und die unter den Operatoren Komposition, primitiver Rekursion und μ -Rekursion abgeschlossen ist.

Diese Klasse von Funktionen ist Turing-vollständig. Dazu muss gezeigt werden, wie man mit einer RAM-Maschine alle partiell rekursiven Funktionen berechnen kann und wie man mit partiell rekursiven Funktionen RAM-Maschinen simulieren kann. Die erste Richtung ist dabei einfacher, da die Programmierung von RAM-Maschinen intuitiver ist. Die Basisfunktionen stellen kein Problem dar. Bei der Komposition werden die Programmfragmente einfach nur aneinander gefügt, es müssen aber gegebenenfalls Register gesichert werden. Die primitive Rekursion lässt sich mithilfe einer Schleife und eines Stacks realisieren. Für die μ -Rekursion ist lediglich eine Schleife durchzuführen, die das minimale y sucht.

Die Rückrichtung ist nicht ganz so einfach. Der Trick besteht hier darin, dass man die Register und den Programmtext selber mithilfe der Paarfunktion jeweils in einer Zahl kodiert. Die Paarfunktion ist eine invertierbare Funktion, die aus zwei beliebig großen natürlichen Zahlen eine natürliche Zahl macht. Man kann beliebig viele natürliche Zahlen in einer Zahl kodieren, wenn man die Funktion mehrmals hintereinander anwendet. Mithilfe der Paarfunktion ist es dann möglich, mit partiell rekursiven Funktionen eine universelle RAM-Maschine zu konstruieren.

Die Idee des Beweises sollte jetzt klar sein; die Details finden sich in [Smi94].

4 Erzeugen von partiell rekursiven Funktionen mit C++ Templates

Die Grundlagen sind gelegt, es ist klar wie C++ Templates und partiell rekursive Funktionen funktionieren. Und es ist klar, dass mit partiell rekursiven Funktionen genau dasselbe berechnet werden kann wie mit Turingmaschinen. Nun bleibt also nur noch zu zeigen, wie man schon beim Kompilieren von C++ Programmen partiell rekursive Funktionen berechnen kann.

Erwin Unruh war 1994 der erste, der auf die Idee kam C++ Templates zum Berechnen von Primzahlen während des Übersetzens zu benutzen. Das ursprüngliche Programm befindet sich auf seiner Internetseite[Unr02] und benutzt neben Templates auch Aufzählungen (enum), um einigermaßen komfortabel mit Zahlen rechnen zu können. Martin Böhme und Bodo Manthey verzichteten auf diesen Komfort und benutzen eine andere Methode, um beliebig große Zahlen konstruieren zu können. Diese Methode wird nun vorgestellt.

4.1 Zahlen und Funktionen

Zunächst benötigen wir eine Darstellung der Null.

```
struct zero { };
```

Der Nachfolger einer Zahl wird mit dem folgenden Template definiert.

```
template<class T> struct suc {
    typedef T pre;
};
```

Templates kann man rekursiv aufeinander anwenden, Eins ist Nachfolger der Null und damit

```
suc<zero>
```

Die Zahl Zwei ist Nachfolger der Eins

```
suc<suc<zero> >
```

und so weiter, wenn die Rekursionstiefe des C++ Übersetzers unendlich groß ist, lassen sich auf diese Weise beliebig große Zahlen darstellen. Ein wichtiges Merkmal des Templates suc ist, dass man damit auch den Vorgänger einer Zahl ermitteln kann. Dieses werden wir zu einem späteren Zeitpunkt noch benötigen. Der Vorgänger von Zwei ist

```
suc<suc<zero> >::pre
```

Der Übersetzer wandelt diesen Typ in den Typ suc<zero> um, der die Zahl Eins darstellt.

Jetzt benötigen wir noch so etwas wie Funktionen, die solche Zahlen als Parameter bekommen und die einen Funktionswert zurückliefern. Wiederum bieten sich Templates an. Die Argumente an die Funktion sind dann die Argumente, die an das Template übergeben werden. Der Funktionswert ist dann der Typ der im Template verschachtelten Typdefinition val. Der Name val ist dabei natürlich frei wählbar, sollte aber fest gewählt werden, damit alle Funktionsaufrufe konsistent sind. Die folgende Funktion addiert Eins auf das übergebene Funktionsargument.

```
template<class X1> struct Suc {
    typedef suc<X1> val;
};
```

Wenn man also die Zahl Eins an diese Funktion wie folgt übergibt

```
Suc<suc<zero> >
```

wird Suc in der Typdefinition vor das übergebene Argument ein suc davor hängen, damit hat man also Eins auf Eins addiert und erwartungsgemäß haben diese beiden Variablen den selben Typ:

```
Suc<suc<zero> >::val variable1;
suc<suc<zero> > variable2;
```

Jetzt möchte man natürlich dieses Ergebnis noch ausgeben. Die Ausgaben beim Kompilervorgang beschränken sich allerdings auf Warnungen und Fehlermeldungen. Naheliegenderweise sollten wir also nun versuchen, eine Fehlermeldung zu erzeugen, in der das Ergebnis vorkommt, also in der der Typ suc<suc<zero> > auftaucht. Dies lässt sich zum Beispiel dadurch erreichen, dass man eine Variable mit dem Ergebnistyp der Funktion anlegt und versucht diese in einen falschen Typ umzuwandeln.

```
int main() {
    Suc<suc<zero> >::val tmp;
    return (int) tmp;
}
```

Normalerweise erzeugt der Übersetzer daraufhin eine Fehlermeldung, die in etwa so lautet:

```
error: 'struct suc<suc<zero> >' used
       where a 'int' was expected
```

Doch auch wenn der Übersetzer nur eine unspezifische Meldung ausgibt, muss er intern die Funktion berechnen. Der Übersetzer muss schließlich die Größe eines Typs kennen. Wenn man also dafür sorgt, dass der Nachfolger eines Typs T (suc<T>) größer als T selber ist, könnte man das Ergebnis der Berechnung später aus der ausführbaren Datei extrahieren.

4.2 Basisfunktionen

Um zu zeigen, dass man auf diese Weise alle partiell rekursiven Funktionen berechnen kann, müssen die Basisfunktionen existieren und die drei Operatoren Komposition, primitive Rekursion und μ -Rekursion müssen auf diese anwendbar sein. Die Basisfunktionen sind am einfachsten umzusetzen. Die Nullfunktion liefert einfach den Wert Null zurück:

```
template<> struct Zero {
    typedef zero val;
};
```

Die Nachfolgerfunktion Suc wurde weiter oben bereits definiert und die Projektionsfunktion liefert einfach das gewünschte Argument als Funktionswert zurück:

```

1  template<class X1> struct Addition_G {
2      typedef X1 val;
3  };
4
5  template<class X1, class X2, class X3> struct Addition_H {
6      typedef suc<X3> val;
7  };
8
9  template <class X, class Y> struct Addition;
10 template<class X> struct Addition<X, zero> {
11     typedef typename Addition_G<X>::val val;
12 };
13
14 template<class X, class Y> struct Addition {
15     typedef typename Addition_H<X, typename Y::pre,
16         typename Addition<X,typename Y::pre >::val >::val val;
17 };

```

Abbildung 2: Umsetzung der Addition mit C++ Templates

```

template<class X1, ...,class XN> struct P {
    typedef XJ val;
};

```

Ebenfalls sehr einfach ist die Komposition von Funktionen. Zur Vereinfachung werden von nun an, alle Funktionen nur mit einem Argument aufgeschrieben, die Verallgemeinerung auf mehr Argumente ist jedoch leicht möglich.

```

template<class X> struct F {
    typedef typename H<
        typename G1<X>::val,
        ...
        typename GN<X>::val>::val val;
};

```

Diese allgemeine Form schreckt womöglich ab, im Prinzip ist es aber so, dass man Funktionswerte bzw. Templates einfach an den Stellen einsetzen kann, an denen man sie benötigt.

4.3 Primitive Rekursion

Die primitive Rekursion in die Template Schreibweise umzusetzen ist ein wenig komplizierter, aber eigentlich auch sehr nahe liegend. Hier ist es nötig das Template einmal für den Fall $y = 0$ und für den Fall $y > 0$ zu spezialisieren. Daher zunächst die *forward* Deklaration für den Übersetzer

```
template <class X, class Y> struct F;
```

Im Fall $y = 0$ wird nun G aufgerufen:

```

template<class X> struct F<X, zero> {
    typedef typename G<X>::val val;
};

```

Für den Fall $y > 0$ benötigen wir die Tatsache, dass man beim Template `suc` auch den Vorgänger ermitteln konnte, da wir anstatt der ursprünglichen Funktion

$$f(x_1, \dots, x_n, y + 1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$$

die äquivalente Funktion

$$f'(x_1, \dots, x_n, y) = h(x_1, \dots, x_n, y - 1, f(x_1, \dots, x_n, y - 1))$$

umsetzen wollen. Die zweite Funktion funktioniert normalerweise nicht, da man keine Vorgängerfunktion zur Verfügung hat. Erst mit der primitiven Rekursion kann man dann eine Funktion entwerfen, die den Vorgänger einer Zahl zurückliefert. Die Funktion f' lässt sich auf folgende Art als Template definieren:

```

template<class X, class Y> struct F {
    typedef typename H<X,typename Y::pre,
        typename F<X, typename Y::pre>::val
    >::val val;
};

```

Man sieht, dass H mit den Parametern X, Y-1 und dem Ergebnis des Funktionsaufrufs F<X, Y-1> aufgerufen wird. Als Beispiel wollen wir nun, wie im vorherigen Abschnitt auch, die Addition von zwei Zahlen umsetzen. Das komplette Beispiel findet sich in Abbildung 2. Die Funktion `Addition_G` liefert dabei bloß das Argument zurück. Bei der Funktion `Addition_H` wird der Nachfolger des dritten Argumentes zurückgegeben. Die Zeilen 9-17 entsprechen der oben entwickelten primitiven Rekursion.

```

1 // Deklaration
2 template<template<class U,class V> class P, class X, class Y, class Q> struct Mu;
3
4 // Fall für P(X, Y) = 0
5 template<template<class U,class V> class P, class X, class Y> struct Mu<P,X,Y,zero> {
6     //liefere Y-1 zurück
7     typedef typename Y::pre min;
8 };
9
10 // Fall für P(X,Y) > 0
11 template<template<class U, class V> class P, class X, class Y, class Q> struct Mu {
12     //liefere Mu(P, X, Y+1, P(X, Y)) zurück
13     typedef typename Mu<P, X, suc<Y>, typename P<X,Y>::val>::min min;
14 };

```

Abbildung 3: μ -Rekursion

4.4 μ -Rekursion

Der letzte Operator, der noch fehlt, ist die μ -Rekursion. Bei der μ -Rekursion geht es darum das minimale y zu suchen, für das ein Prädikat den Wert Null zurückliefert. Außerdem müssen aber auch alle Werte getestet werden, die kleiner als y sind. Daher müssen wir nun eine Schleife konstruieren, die alle Zahlen bei Null beginnend mit dem Prädikat p testet, solange bis p Null zurückliefert. Es ist also die folgende Funktion zu programmieren

$$mu(x, h, y, p) = \begin{cases} y - 1 & \text{für } p = 0 \\ mu(x, p, y + 1, p(x, y)) & \text{sonst} \end{cases}$$

Mit dem Prädikat $p(x_1, \dots, x_n, y)$ testen wir die Schleifenbedingung im Prinzip immer erst eine Rekursionstiefe später. Das ist der Grund dafür, dass $y - 1$ im Fall $p = 0$ zurückgegeben wird. Weiterhin ist es so, dass dadurch die Funktion mit $p > 0$ aufgerufen werden muss, da der eigentliche Test für den Durchlauf $y = 0$ erst bei $y = 1$ stattfindet. Es ergibt sich als Aufruf für die μ -Rekursion

$$f(x) = \mu y [p(x_1, \dots, x_n, y) = 0] = mu(x_1, \dots, x_n, p, 0, 1)$$

Die Umsetzung in Templates erfolgt sehr ähnlich zur primitiven Rekursion. In Abbildung 3 ist die komplette Definition zu sehen. Der einzige Unterschied besteht darin, dass an den Platz der herunter zählenden Variable Y hier die Ausführung des Prädikates P gerückt ist (Zeile 13), das man als Parameter dem Template übergeben hat.

Um die μ -Rekursion aufzurufen, deklariert man zunächst das Prädikat P

```

template<class U, class V> struct Endlos {
    typedef suc<zero> val;
};

```

und ruft die μ -Rekursion mit diesem auf

```

Mu<Endlos, zero, zero, suc<zero> >::min tmp;
return (int) tmp;

```

In diesem Beispiel landet der Übersetzer in einer Endlosschleife, da H niemals Null zurückliefert, die Templates werden also unendlich tief verschachtelt.

5 Zusammenfassung und Fazit

Wir haben gesehen, wie es mithilfe der Spezialisierung von Templates und Typdefinitionen möglich ist, partiell rekursive Funktionen zu berechnen. Das Ergebnis der Berechnung wurde dabei als Fehlermeldung, während des Übersetzens ausgegeben.

Eine andere Methode zu zeigen, dass der C++ Template Mechanismus Turing-vollständig ist, findet sich in [Vel03]. Todd Veldhuizen konstruiert darin mit Templates eine Turingmaschine, deren Konfigurationen man bei der Übersetzung als Fehlermeldungen ausgegeben bekommt.

Lässt sich dieses Konzept nun für irgendwelche praktischen Sachen gebrauchen? Ja, durch Todd Veldhuizen initiiert [Vel95] fand die Template Metaprogrammierung rege Verbreitung in der C++ Gemeinde. Mit der Template Metaprogrammierung lassen sich Algorithmen schon während der Übersetzung auf die Parameter spezialisieren, mit denen sie später aufgerufen werden. Damit lassen sich zum Teil beträchtliche Geschwindigkeitsvorteile erreichen. Weiterhin lassen sich schon während des Übersetzens Konstanten berechnen, es ist schließlich einfacher, eine Variable mit `FACTORIAL<5>::value` zu initialisieren, als mit `5*4*3*2` vor allem für größere Werte von n . Mittlerweile existiert auch die Library `MPL[GA02]`, die alle Konzepte der Metaprogrammierung zur Verfügung stellt. Aus einem Scherz wurde also bitterer Ernst.

Literatur

- [BM03] BÖHME, Martin ; MANTHEY, Bodo: The Computational Power of Compiling C++. In: *Bulletin of the European Association for Theoretical Computer Science* 81 (2003), Oktober, S. 264–270
- [GA02] GURTOVOYI, Aleksey ; ABRAHAMS, David: *The Boost C++ Metaprogramming Library*. <http://www.mywikinet.com/mpl/>. Version: 2002
- [Int03] INTERNATIONAL STANDARD ISO/IEC 14882: *Programming Languages – C++*. 2. ISO/IEC: Geneva, 2003
- [Smi94] SMITH, Carl H.: *A recursive introduction to the theory of computation*. Springer, 1994
- [Str97] STROUSTRUP, Bjarne: *The C++ Programming Language*. 2. Addison-Wesley, 1997
- [Unr02] UNRUH, Erwin: *Template Metaprogramming*. <http://www.erwin-unruh.de/meta.html>. Version: 2002
- [Vel95] VELDHUIZEN, Todd: Using C++ template metaprograms. In: *C++ Report* 7 (1995), Mai, Nr. 4, S. 36–43. – Reprinted in *C++ Gems*, ed. Stanley Lippman. – ISSN 1040–6042
- [Vel03] VELDHUIZEN, Todd: C++ Templates are Turing complete. (2003). <http://osl.iu.edu/~tveldhui/papers/2003/turing.ps>
- [You67] YOUNGER, Daniel H.: Recognition and parsing of context-free languages in time n^3 . In: *Information and Control* 10 (1967), Februar, Nr. 2, S. 189–208
- [Zie04] ZIEGLER, Martin: *Advanced Computability Theory*. (2004). <http://www.upb.de/cs/ag-madh/vorl/Computability/SCRIPT.ps.gz>