SkelML — Ein Skeleton-basierter parallelisierender Compiler für ML

Jürgen Nickelsen

Ausarbeitung zum Referat im Seminar
„Implementierung paralleler funktionaler Programmiersprachen“
am 16. Mai 1994

An der Herriot-Watt-University in Edinburgh arbeitet Tore A. Bratvold am Department of Computing and Electrical Engineering an einem parallelisierenden Compiler für ML auf der Basis von Skeletons. Dieser Compiler erzeugt Code für ein Transputer-Netzwerk.

Meinem Referat (und damit dieser Ausarbeitung) liegen zwei Veröffentlichungen von Tore A. Bratvold und persönliche Korrespondenz mit ihm per E-Mail zugrunde.

1.0 Einleitung

Programmiersprachen mit Konstrukten zur expliziten Nutzung von Parallelrechnern bieten der Programmiererin / dem Programmierer[1] die Möglichkeit, Parallelisierung sehr gezielt einzusetzen. Die Entwicklung größerer paralleler Programme mit diesen Mitteln ist jedoch eine lästige und fehleranfällige Arbeit. In den letzten Jahren wurden Programmiersysteme entwickelt, die bestimmte Muster von parallelisierbaren Berechnungen, *Skeletons*, als direkt benutzbare Funktionen zur Verfügung stellen und damit eine von der expliziten Parallelisierung abstrahierende Arbeitsweise ermöglichen.

1. Um weitere sprachliche Verrenkungen zu vermeiden, werde ich im folgenden nur die weibliche Form benutzen, wo sowohl die männliche als auch die weibliche Form gemeint ist.

Die Programmiererin hat damit eine bestimmte Anzahl von Konstruktoren zur Verfügung, die jeweils zu einem bestimmten Skeleton gehören. Durch die Verwendung eines solchen Konstruktors kann Parallelisierung ausgenutzt werden, ohne daß deren Low-Level-Details Beachtung geschenkt werden muß. Idealerweise sollte die Programmiererin nur die *Bedeutung* des Programms spezifizieren, wobei es dem Compiler überlassen bleibt, das genaue *Verhalten* auf der Zielmaschine zu bestimmen. Damit dieser Ansatz zum Erfolg führt, ist es wichtig, daß der Compiler diese Aufgabe nicht wesentlich schlechter erledigt als eine versierte Programmiererin.

2.0 Skeletons

Skeletons sind, wie im vorhergehenden Referat besprochen, Programmstrukturen mit bekannter Parallelisierbarkeit, die direkt in Programmen verwendet werden können bzw. schon verwendet werden. In funktionalen Programmen sind es typischerweise Funktionen höherer Ordnung (*Higher Order Functions*, HOFs), die als Skeletons zur Parallelisierung verwendet werden können, sowie die Pipeline.

2.1 Von SkelML unterstützte Skeletons

map

```
fun map f [] = [] |
    map f (h::t) = f h :: map f t;
```

filter

```
fun filter p [] = [] |
    filter p (h::t) = if p h then
                          h::(filter p t)
                      else
                          filter p t;
```

fold

```
fun fold (f:‘a->‘a->‘a) a [] = a |
    fold f a (h::t) = f h (fold f a t);
```

filtermap

```
(filter p) o (map f)
```

mapfilter

```
(map f) o (filter p)
```

foldmap

```
(fold f a) o (map g)
```

Zusätzlich wird die Pipeline als Skeleton unterstützt. In funktionalen Sprachen ist das die Verkettung von Funktionen, also

auch wieder eine HOF, die in ML durch den Operator „o“ ausgedrückt wird.

2.2 Vorteile von Skeletons

Mit Skeletons kann implizite Parallelität genutzt werden, die in Programmen auch ohne Zutun der Programmiererin dort schon enthalten ist, wo die entsprechenden HOFs benutzt werden. Die Programmierung braucht strenggenommen überhaupt nicht umgestellt zu werden; auch bestehende (sequentielle) Programme können so von der Parallelisierung profitieren. Es ist aber andererseits möglich, durch eine forcierte Nutzung der Skeletons die Parallelisierbarkeit von Programmen zu erhöhen.

Da HOFs in der funktionalen Programmierung ohnehin gerne und viel benutzt werden, ist eine Umstellung auf eine parallele Variante einer funktionalen Programmiersprache sehr leicht. Der bisher gewohnte Programmierstil braucht zunächst überhaupt nicht geändert zu werden; mit der Zeit ist es dann sinnvoll, zu einer stärkeren Benutzung von HOFs überzugehen.

Die Benutzung von Skeletons ist architekturunabhängig, weil innerhalb des Quellprogramms keine explizite Parallelisierung erfolgt. Daher muß ein solches Programm die Zielarchitektur nicht kennen – die Abbildung auf die konkrete Maschine ist ausschließlich Sache des Compilers.

Mit Skeletons ist es möglich, sowohl Kontroll- als auch Datenparallelität zu unterstützen. In welchem Ausmaß das jeweils geschieht, hängt von den zur Verfügung gestellten Skeletons ab.

2.3 Nachteile von Skeletons

Von der Anzahl möglicher Skeletons und der Häufigkeit ihrer Verwendung hängt es ab, inwieweit ein Programm damit parallelisierbar ist. Wir sehen hier am Beispiel der von SkelML zur Verfügung gestellten Skeletons, daß diese Auswahl noch recht begrenzt ist – dementsprechend begrenzt ist auch die damit erreichte Parallelisierung funktionaler Programme.

3.0 Die Prozessor-Farm

Tore Bratvold benutzt für die zugrundeliegende Maschinenarchitektur das Modell der Prozessor-Farm. Dabei nimmt ein designierter Prozessor, der *Farmer*, den Eingabe-Datenstrom entgegen und verteilt davon zu bearbeitende Datenportionen auf die anderen Prozessoren, die *Worker*, die die Resultate der Berechnungen wieder an den Farmer zurückgeben.

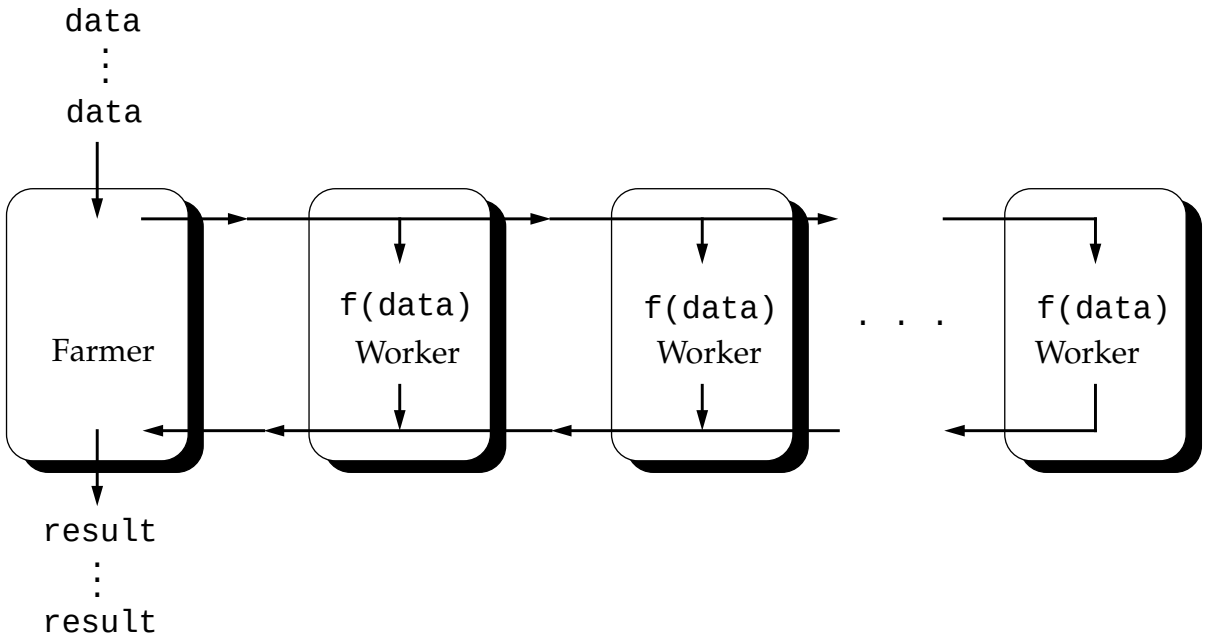


ABBILDUNG 1. Prozessor-Farm

Die Prozessor-Farm ist ein sehr allgemeines Modell für parallele Berechnung, mit dem eine Reihe von HOFs effizient implementiert werden können. Ein attraktives Merkmal der Farm ist die Möglichkeit, die Last gleichmäßig auf die Worker verteilen zu können, wenn die Anzahl der zu bearbeitenden Datenpakete hoch genug ist.

Die Performance der Prozessor-Farm hängt einerseits von der Arbeitsgeschwindigkeit der Worker, andererseits vom Kommunikationsaufwand ab, der sich aus den Kosten zur Datenverteilung und den Kosten zur Rekombination der Daten zusammensetzt.

Die Prozessoren in einem Prozessor-Netzwerk können in mehrere Prozessor-Farmen aufgeteilt werden; dabei entspricht jede

einzelne Farm einer Parallelisierungseinheit, also hier einer zu berechnenden HOF.

4.0 Arbeitsweise des Compilers

4.1 Möglichkeiten zur Parallelisierung

Für die automatische Parallelisierung von Programmen gibt es im wesentlichen drei verschiedene Ansätze:

- Das Programm wird statisch auf zu parallelisierende Stellen analysiert. Dabei können nur wenige Aussagen über die Nützlichkeit der Parallelisierung an einer bestimmten Stelle gemacht werden.
- Die Programmiererin macht Anmerkungen zum Programm, in denen zu parallelisierende Stellen angegeben werden. Ein Nachteil dieser Methode ist, daß diese Anmerkungen mehr oder auch weniger sinnvoll sein können – die Lastverteilung im tatsächlichen Einsatz kann anders sein als ursprünglich angenommen.
- Anhand von Laufzeitanalysen werden die Stellen im Programm ermittelt, an denen eine Parallelisierung den maximalen Performancegewinn bringt.

4.2 Realisierung in SkelML

SkelML benutzt den dritten Ansatz zur Parallelisierung und Lastverteilung auf die einzelnen Prozessoren. Dabei geht der Compiler in folgenden Schritten vor:

1. Das Frontend von SkelML besteht aus einem ML-Interpreter, der eine (große) Untermenge von SML versteht. Er ist im „Look and Feel“ an die SML-Compiler aus Edinburgh und New Jersey angelehnt. Dieser Interpreter kann für die Programmentwicklung unabhängig benutzt werden, aber er enthält zusätzlich eine Anzahl von Kommandos, um den parallelisierenden Compiler aufzurufen und zu steuern. Dieser Interpreter erstellt den mit Typinformation versehenen Graphen des Programms und gibt ihn weiter an den Profiler.
2. Der Profiler nimmt den vom Frontend erzeugten Programmgraphen und erstellt eine Laufzeitanalyse mit Hilfe von beispielhaften Eingabedaten. Dabei wird eine detaillierte Statistik über das Laufzeitverhalten des Programms erstellt. Grundlage für diese Statistik sind dabei die angenommenen

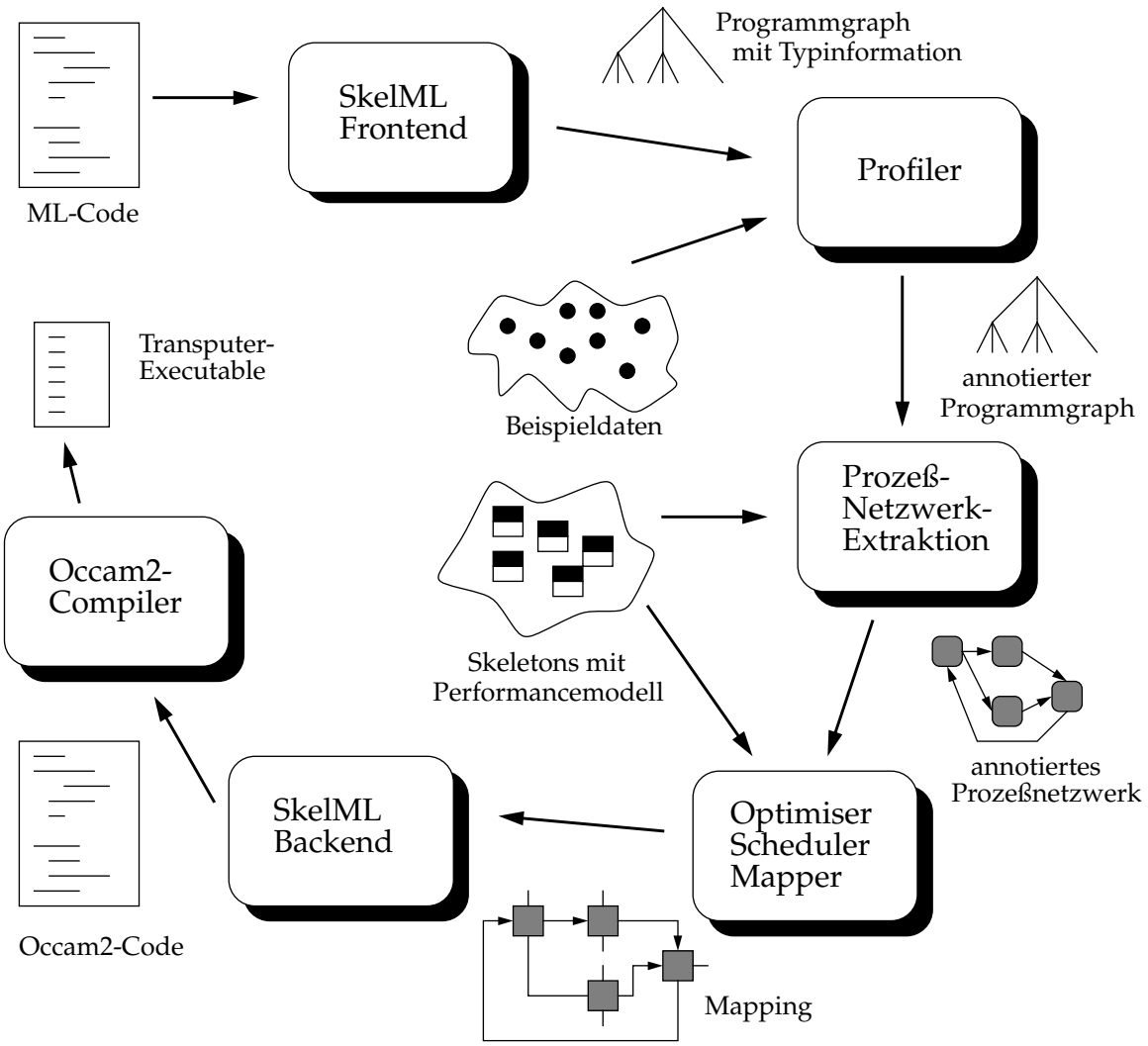


ABBILDUNG 2. Überblick über die Phasen des Compilers

Ausführungszeiten des Programms auf der Zielmaschine; diese sind für die einzelnen Sprachelemente von ML dem Profiler bekannt.

3. Zu dem so mit Informationen versehenen Programmgraphen wird in der nächsten Phase ein mit entsprechenden Informationen zum Laufzeitverhalten versehenes Prozeß-Netzwerk erzeugt. Dabei gehen schon Informationen über die zur Verfügung stehenden Skeletons und ihr Laufzeitverhalten ein.
4. Aus dem virtuellen Prozeß-Netzwerk wird in der Optimierungsphase das tatsächliche Mapping auf das Prozessor-Netzwerk gebildet. Hier gehen wieder die Laufzeitinforma-

tionen über das Programm und die Performance-Modelle der Skeletons ein.

5. Aus dem Mapping wird im SkelML-Backend Occam2-Code erzeugt.
6. Ein Occam2-Compiler übersetzt diesen Code in ein ausführbares Programm für das Transputer-Netzwerk.

Die wesentlichen Schritte der Parallelisierung lassen sich kurz so zusammenfassen:

1. Identifizieren der HOFs im Programmtext,
2. Bestimmen der nützlichen Parallelität anhand von Laufzeitanalysen,
3. Lastverteilung für das Prozessornetzwerk.

4.3 Mapping eines Prozeß-Netzwerks

Als ein Beispiel für das Mapping eines Prozeß-Netzwerks auf das tatsächliche Prozessor-Netz betrachten wir ein Raytracer-Programm mit der folgenden Struktur:

```
datatype Impact = IMPACT of real * int
                | NOIMPACT;
(* IMPACT(dist, objno) gibt an, daß der Sehstrahl
   in der Entfernung dist auf das Objekt objno
   trifft. *)

fun FirstImpact objects ray = ... ;
(* Findet den ersten Impact, oder NOIMPACT,
   zwischen dem Strahl und den Objekten. *)

fun FindImpacts rays objects =
          map (FirstImpact objects) rays;
(* Findet alle Impacts von rays auf objects. *)

fun GenerateRays (Detail, X, Y, Z) = ... ;
(* Generiert Detail * Detail Strahlen vom
   Blickpunkt X, Y, Z in Richtung des Ursprungs. *)
```

```
fun top (Detail, XView, YView, ZView) Scene =
     FindImpacts (GenerateRays
                    (Detail, XView,
                     YView, ZView)) Scene;
(* Top-Level-Funktion. *)
```

Die Strahlen werden vom durch `XView, YView, ZView` gegebenen Blickpunkt verfolgt. Der Integer-Wert `Detail` gibt an, daß `Detail * Detail` Strahlen erzeugt werden. Der Parameter `Scene` enthält die Liste der Objekte, die auf Impacts untersucht werden. Das Ergebnis des Programms ist eine Liste von Impact-Informationen für jeden Strahl. Diese Information besteht aus der Entfernung des Impacts und des getroffenen Objekts oder aus dem Konstruktor `NOIMPACT`, wenn kein Objekt getroffen wurde.

Wird dieses Programm SkelML als Eingabe gegeben, wird das `map` in `FindImpacts` als parallelisierbar erkannt. SkelML generiert dann ein entsprechendes Prozeß-Netzwerk:

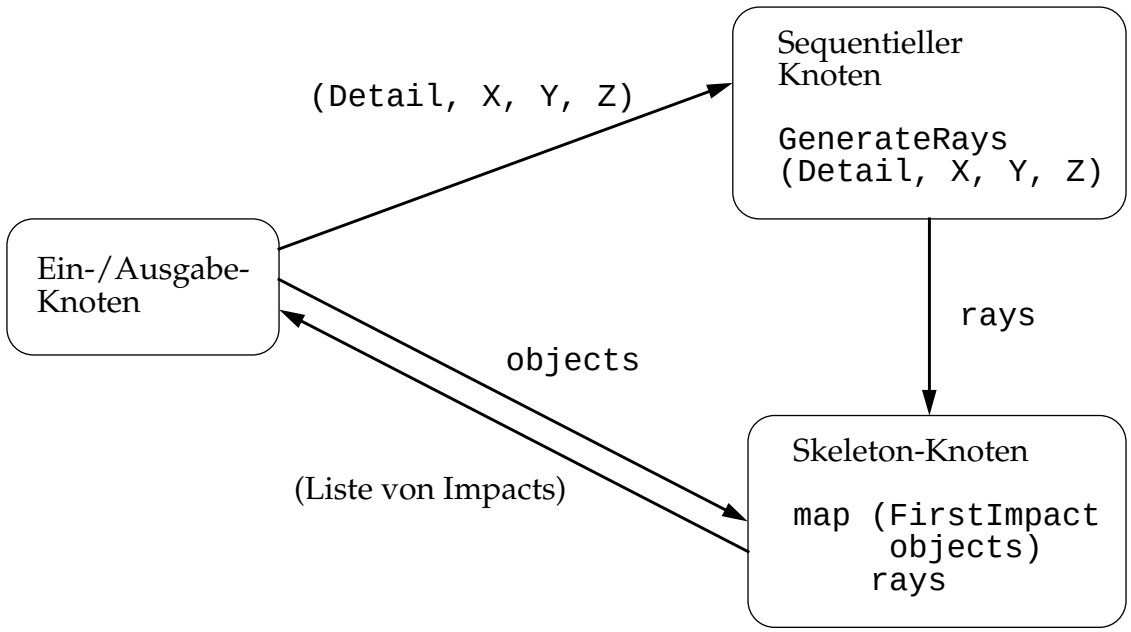


ABBILDUNG 3. Prozeß-Netzwerk

Das daraus erzeugte Mapping auf die Prozessoren kann dann so aussehen:

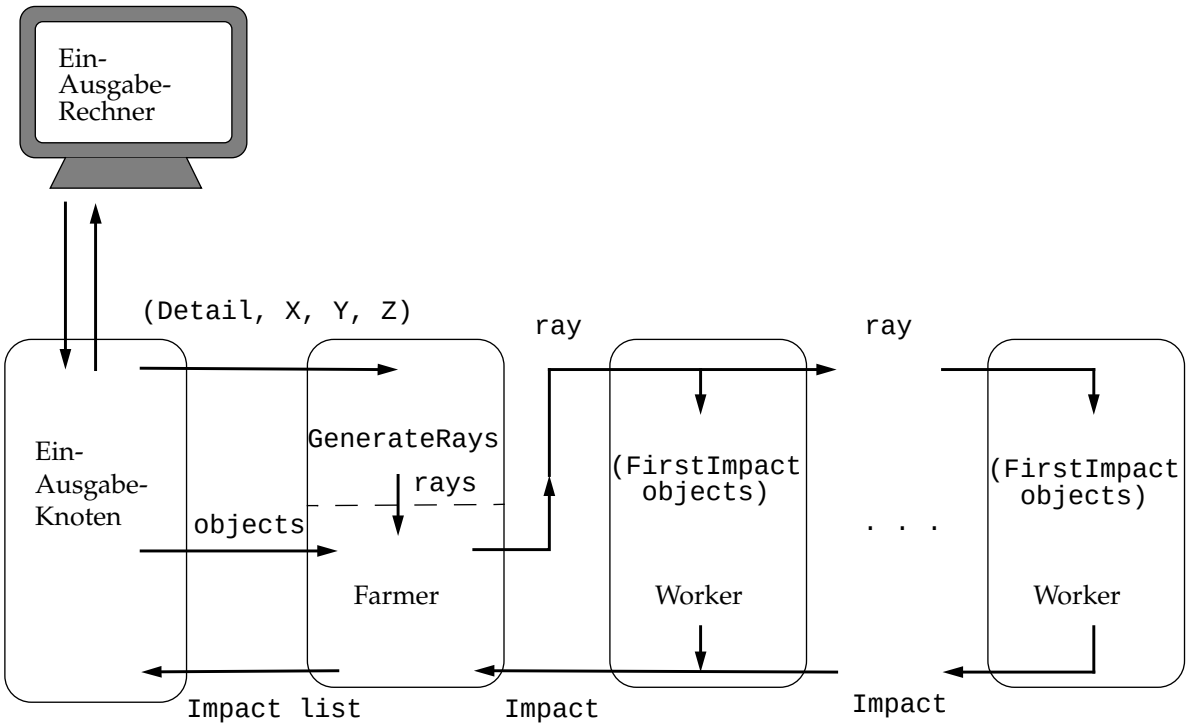


ABBILDUNG 4. Mapping auf das Prozessor-Netz

4.4 Transformationen

Um eine günstigere Lastverteilung zu erreichen, führt SkelML zusätzlich Transformationen auf dem Programm durch. Dabei werden zur Zeit folgende Transformationen angewendet:

1. `map f (map g l)`
   `→ map (fn x => g(f x)) l;`
2. `map f (filter p l)`
   `→ mapfilter f p l;`
3. `filter p (map f l)`
   `→ filtermap p f l;`
4. `fold f a (map g l)`
   `→ foldmap f a g l;`
5. `filter p2 (filter p1 l)`
   `→ filter (fn x => (p1 x)`
   `andalso (p2 x)) l;`
6. `filter p1 (filter p2 l)`
   `→ filter p2 (filter p1 l);`

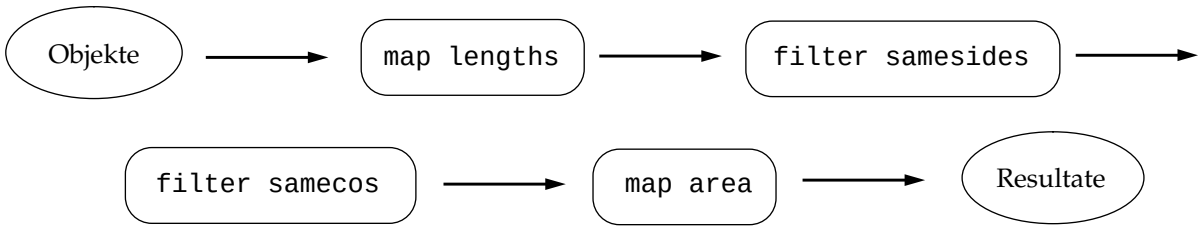
Die letzte dieser Transformationen ist nicht semantikerhaltend; wenn beispielsweise das Prädikat `p2` aus der Liste `l` die von Null verschiedenen Zahlen herausfiltert und das Prädikat `p1` eine Division durch das Listenelement beinhaltet, wird die Semantik durch die Vertauschung der Filteroperationen verändert. Tore Bratvold sagt selbst, daß diese Transformation deshalb nicht automatisch durchgeführt werden kann.

Zu den Transformationen hier ein Beispiel: Berechnet werden soll eine Funktion

```
fun program objects =
    map area (filter samecos
                     (filter samesides
                             (map lengths
                                  objects)));
```

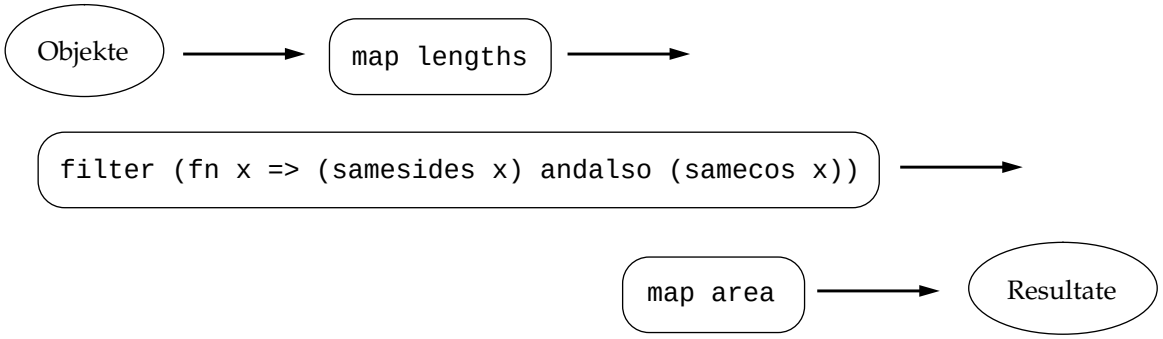
die die Flächen aller regulären Objekte in der Eingabeliste zurückgibt. Dabei berechnet `lengths` die Seitenlängen eines zweidimensionalen Objekts; `samesides` ist wahr, wenn alle Seiten gleich lang sind; `samecos` ist wahr, wenn alle Winkel gleich sind; `area` berechnet die Fläche eines Objekts.

Eine einfache Implementierung dieser Funktion wäre eine Pipeline von vier Prozessor-Farmen:

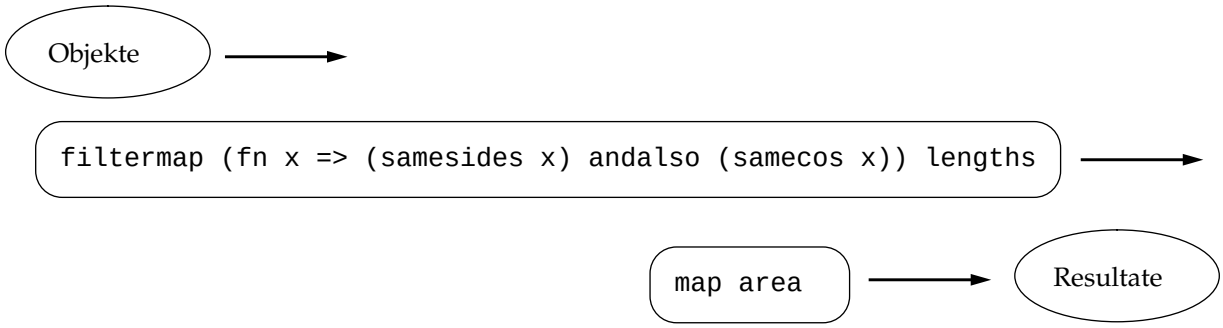


Das ist aber nicht besonders effizient, weil die Funktion `area` viel rechenaufwendiger ist als `length`, `samesides` und

`samecos`. Deshalb wenden wir Transformation 5 an, um die Filterfunktionen zusammenzufassen:



Dann wenden wir Transformation 3 an, um `map lengths` und `filter` zusammenzufassen:



Diese Pipeline aus zwei Farmen hat schließlich eine bessere Performance als die ursprüngliche Pipeline aus vier Farmen.

5.0 Stand der Implementierung

Zur Zeit gibt es einen in C++ implementierten Prototyp von SkelML, der auf einer Sun-SPARC-Maschine läuft. Gegenüber den zur Zeit des Referats verfügbaren Informationen hat sich die Situation inzwischen etwas geändert.

Dieser Compiler macht eine brauchbare Performance-Analyse und entsprechende Modellierung mit Skeletons, sequentiellem Code und der Kommunikation. Er führt ein automatisches Mapping der Prozesse auf die Prozessoren durch und benutzt Transformationen, um Pipelines zu optimieren.

6.0 Zusammenfassung

SkelML ist ein Compiler für ML, der für ein Transputer-Netzwerk auf Basis von Skeletons parallelen Code erzeugt. Zur Performance-Optimierung wird eine Laufzeit-Analyse des Programms mit Beispieldaten benutzt, um für die Parallelisierung gut geeignete Stellen des Programms auszuwählen. Um die Lastverteilung auf die Prozessoren zu verbessern, werden Transformationen auf dem Programm durchgeführt. In die Optimierung gehen Kenntnisse über das Laufzeitverhalten der Skeletons inklusive des Kommunikationsaufwands mit ein.

Der Erfolg dieser Vorgehensweise ist sehr stark von der Qualität der Beispieldaten abhängig. Wenn diese Daten den Daten der späteren Anwendung nicht entsprechen, kann die Performance aufgrund einer inadäquaten Abbildung auf die Zielmaschine schlecht sein.

Trotz dieser Schwächen scheint die Nutzung von Skeletons als Hilfsmittel zur Parallelisierung von Programmen attraktiv, weil die Programmiererin nicht mit den Details der parallelen Programmierung belastet wird und in gewohnter Weise auf einem hohen Abstraktionsniveau arbeiten kann. Durch eine sorgfältige Auswahl der Beispieldaten muß ein Einbrechen der Performance in der Anwendung der Programme vermieden werden.

Quellen

Tore A. Bratvold: Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, September 1992, pp. 213 – 216. RWTH Aachen / Aachener Informatik-Berichte Nr. 92-19.

Tore A. Bratvold: A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the 5th International Workshop on Implementation of Functional Languages*, Nijmegen, Niederlande, September 1993, pp. 23 – 33.

Tore A. Bratvold, persönliche Kommunikation (E-Mail). April – Juli 1994.