Eine typisierte, rein funktionale Modulsprache für das Programmieren-im-Großen

Erweiterte Zusammenfassung

Franz-Josef Grosch
Abt. Softwaretechnologie, TU Braunschweig

1 Einleitung

Die Bausteine des Programmierens-im-Großen, die Module, sind Sammmlungen einfacher Programmkomponenten: Werte, Funktionen und Typen in funktionalen Sprachen; zusätzlich Prozeduren und globale Variablen in imperativen Sprachen sowie Klassendefinitionen in objekt-orientierten Sprachen. Parametrisierte Module sind Funktionen über Modulen, deren Eingabe- und Ausgabeschnittstellen Importe und Exporte beschreiben. Unabhängig vom internen Charakter einzelner Module können große Systeme durch Anwendung dieser Funktionen schichtweise konstruiert werden. Die Anwendung auf existierende Module erzeugt neue Module. Betrachtet man ferner Ein- und Ausgabeschnittstellen als Typen, so erkennt man: Programmieren-im-Großen ist typisiertes funktionales Programmieren [1].

Ansatzweise wird typisiertes Programmieren-im-Großen in vielen Sprachen unterstützt. Richtungsweisend war Modula-2: Definitionsmodule beschreiben Schnittstellen (Typen) und Implementierungsmodule enthalten die Realisierung; die Systemstruktur ist allerdings statisch. Das Modulsystem von Standard ML (SML) geht wesentlich weiter, es ist tatsächlich eine eigene typisierte Modulsprache mit einfachen Modulen (Strukturen), Schnittstellen (Signaturen) und parametrisierten Modulen (Funktoren). Durch Funktorapplikation werden Strukturen aneinander gebunden. Allerdings ist auch die Modulsprache von SML nicht wirklich funktional: Gleiche Funktorapplikationen erzeugen verschiedene Strukturen (Generativität). Aussagen oder Beweise über die Systemstruktur werden so erschwert und die referentielle Transparenz ist verletzt. Neue Arbeiten erweitern das Modulsystem von SML um Funktoren höherer Ordnung [2], allerdings mit einer *stamp*-basierten operationalen Semantik, die nicht funktional ist. Leroy [3] schlägt eine Variante vor, die zumindest für die bei der Funktorapplikation erzeugten Typen funktionales Verhalten hat.

Wir stellen eine typisierte, rein funktionale Modulsprache vor, die im Unterschied zu SML referentiell transparent ist und softwaretechnische Prinzipien besser unterstützt. Prinzipiell kann jede typisierte Programmiersprache als Implementierungssprache in die Modulsprache eingebettet werden. Das Paradigma der Implementierungssprache – imperativ, funktional oder objekt-orientiert – bestimmt lediglich den Charakter der elementaren Module, während die Modulsprache selbst funktional bleibt. Die Grundlage für die Modulsprache ist ein auf dem λ-Kalkül aufbauender $\lambda\delta$-Modulkalkül, der es erlaubt, Generativität von Typen und Modulen durch ein spezielles Bindungskonstrukt seiteneffektfrei zu beschreiben. Der Kalkül ist mit *dependent types* [4,5,6] typisiert, wobei die Typisierung von den Typen der eingebetteten Implementierungssprache abhängt. Ausdrücke höherer Ordnung, die Bildung von Teilsystemen und die Wiederverwendung ganzer Teilsysteme sind integraler Bestandteil des Sprachkonzepts. Das Ergebnis der Ausführung eines Modulprogramms ist, sofern es sich um einen beobachtbaren Ausdruck handelt, ein System.

In dieser Zusammenfassung werden wir zunächst den Kalkül kurz vorstellen und dann anhand eines Beispiels die Nutzung der Modulsprache demonstrieren. Als Implementierungssprache für elementare Module dient der Kern von SML, die Schnittstellen elementarer Module werden daher auch in Form von SML-Signaturen beschrieben. Das Beispiel soll verdeutlichen, wie der Entwurf von Referenzarchitekturen (Architekturen für eine Produktlinie) und die Erzeugung spezieller Systeme (Produkte) programmiersprachlich unterstützt werden kann. Abschließend wollen wir skizzieren, wie die Einhaltung softwaretechnischer Prinzipien durch die Modulsprache unterstützt wird.

Modulausdrücke

t	$::=$	t_p	Modul-/Typpfad
	$\mid$	$(\lambda\, x{:}T\,.\,t)$	(Funktions-)Abstraktion
	$\mid$	$(t_1\ t_2)$	(Funktions-)Applikation
	$\mid$	$(\delta\, x{:}T\,.\,t)$	(System-)Deklaration
	$\mid$	$(\textbf{bind}\ x\ \textbf{to}\ t_1\ \textbf{in}\ t_2)$	(System-)Verwendung
t_p	$::=$	x	Modul-/Typbezeichner
	$\mid$	l	Modul-/Typlabel
	$\mid$	$t.l$	qualifizierter Modul-/Typname

Schnittstellenausdrücke

T	$::=$	T_s	(einfache) Schnittstelle
	$\mid$	$(\Pi\, x{:}T_1\,.\,T_2)$	Funktionsschnittstelle (*dependent function type*)
	$\mid$	$(T\ t)$	Applikation einer Funktionsschnittstelle
	$\mid$	$(\exists\, x{:}T_1\,.\,T_2)$	Systemschnittstelle (*dependent sum type*)
	$\mid$	$(\textbf{Bind}\ x\ \textbf{To}\ t\ \textbf{In}\ T)$	Verwendung einer Systemschnittstelle
T_s	$::=$	$\textbf{sig}\ D_1, \ldots, D_n\ \textbf{end}$	Modulschnittstelle
	$\mid$	**type**	Typschnittstelle (*kind*)
D	$::=$	$l : T$	opaque Modul-/Typdeklaration
	$\mid$	$l : T = t$	manifeste Modul-/Typdeklaration
	$\mid$	$l : \tau$	Wertdefinition
τ	$::=$	t_p	(qualifizierter) Typbezeichner
	$\mid$	$t_p \to t_p \mid list(t_p) \mid \ldots$	Typen der Implementierungssprache

Reduktionsregeln

$(\lambda\, x{:}T\,.\,t_1)\ t_2$	$\to_t$	$t_1[x := t_2]$
$\textbf{bind}\ x_2\ \textbf{to}\ (\delta\, x_1{:}T\,.\,t_1)\ \textbf{in}\ t_2$	$\to_t$	$\delta\, x_1{:}T\,.\,(t_2[x_2 := t_1])$
$(\Pi\, x{:}T_1\,.\,T_2)\ t$	$\to_T$	$T_2[x := t]$
$\textbf{Bind}\ x_2\ \textbf{To}\ (\delta\, x_1{:}T_1\,.\,t)\ \textbf{In}\ T_2$	$\to_T$	$\exists\, x_1{:}T_1\,.\,(T_2[x_2 := t])$

Abbildung 1: Syntax und Reduktionsregeln des $\lambda\delta$-Modulkalküls

2 Der $\lambda\delta$-Modulkalkül

Die Basis der funktionalen Modulsprache ist, wie üblich, ein λ-Kalkül. Allerdings ist der reine λ-Kalkül ein Kalkül von Kontrollstrukturen, den man als Grundlage einer Programmiersprache um Datenstrukturen erweitert. Üblicherweise wird bei funktionalen Programmiersprachen der λ-Kalkül um vordefinierte Konstruktoren und Operationen erweitert. Für eine Modulsprache muß er um elementare Module als Datenstrukturen erweitert werden. Elementare Module können allerdings nicht als vordefiniert betrachtet werden, da sie von anderen Modulen abhängen können. Aus diesem Grund wird ein zweites Bindungskonstrukt δ und ein zugehöriges Eliminationskonstrukt **bind** eingeführt, daß es erlaubt, den Namen eines „neuen“ Moduls einzuführen bzw. den Sichtbarkeitsbereich eines δ-gebundenen Namens zu erweitern. Abbildung 1 zeigt die Syntax und die Reduktionsregeln des $\lambda\delta$-Modulkalküls.

Die Werte des $\lambda\delta$-Modulkalküls sind Module und Typen der Implementierungssprache. Funktionsabstraktionen abstrahieren von konkreten („hartverdrahteten“) Importen; durch Funktionsapplikation werden Module aneinander gebunden (*linking*). In der Systemdeklaration wird der

Modulausdrücke	:	`module m = ...`
$\lambda\, x{:}T\,.\,t$	$\simeq$	`connect x : T -> t`
$\delta\, x{:}T\,.\,t$	$\simeq$	`define x : T <- t`
Schnittstellen	:	`interface S = ...`
$\Pi\, x{:}T_1\,.\,T_2$	$\simeq$	`all x : T1 => T2`
$\exists\, x{:}T_1\,.\,T_2$	$\simeq$	`some x : T1 <= T2`

Abbildung 2: Syntax der Modulsprache

Name eines „neuen“ elementaren Moduls, der im Rumpf verwendet werden kann, definiert. Die Systemverwendung erlaubt, den Rumpf einer Systemdeklaration an einen Bezeichner zu binden und diesen weiter zu verwenden. Aus diesem Grund nennen wir den Rumpf einer Systemdeklaration auch „Exportteil“.

Die Reduktionsregeln für Modulausdrücke umfassen die übliche β-Reduktion für Funktionsapplikation sowie eine Regel für die Verwendung von Systemen, bei der der Gültigkeitsbereich des δ-gebundenen Moduls erweitert wird und der Rumpf (Exportteil) unter einem neuen Namen benutzt werden kann. Beide Reduktionsregeln sind unter Berücksichtigung der α-Konversion zu verstehen. Man erkennt: δ-gebundene Bezeichner werden niemals ersetzt, lediglich ihr Gültigkeitsbereich wird verändert. Ausdrücke, die ausschließlich δs enthalten, sind beobachtbar und beschreiben eine konkrete Systemarchitektur.

Der Modulkalkül ist typisiert unter Verwendung der Typen einer Implementierungssprache. Die Typen von Modulausdrücken heißen Schnittstellen. Einfache Schnittstellen sind die Schnittstellen elementarer Module, vergleichbar mit den Definitionen eines Definitionsmoduls in Modula-2 oder einer Signatur in SML. Eine einfache Schnittstelle beschreibt Typen und Werte der Implementierungssprache sowie Submodule, die durch das zugehörige Modul zur Verwendung angeboten (exportiert) werden. Typen und Submodule können sowohl opak (abstrakt) als auch transparent (manifest) exportiert werden. Zusammengesetzte Ausdrücke des Modulkalküls sind mit *dependent types* typisiert. Die Ergebnisschnittstelle einer Funktion hängt vom konkreten Parametermodul ab. Die Exportschnittstelle eines Systems hängt ab von den δ-gebundenen Modulen.

Der Kalkül läßt beliebige Ausdrücke höherer Ordnung zu, mit einer Ausnahme: δ-gebundene Namen müssen immer elementare Module sein. Die wesentliche Eigenschaft des Modulkalküls ist die strenge Normalisierung.

3 Ein Beispiel

In diesem Abschnitt soll ein Beispiel aus Paulsons ML-Buch [7] in die Modulsprache übertragen werden. Die Syntax der Modulsprache ist in Abbildung 2 skizziert, sie ist im wesentlichen equivalent zum Kalkül. Der Kalkül, so wie er bisher vorgestellt wurde, bietet keine Möglichkeit zur Implementierung elementarer Module. Tatsächlich gehört zu jedem δ-gebundenen Bezeichner eine Implementierung, die syntaktisch nach der Signatur angegeben werden kann und genau wie die Signatur auf alle hier sichtbaren Bezeichner zugreifen darf. In der Beispielarchitektur ist eine Implementierung für das Modul `LamKey` angegeben (Abbildung 3).

Im Beispiel soll aus einer Referenzarchitektur für Syntaxanalyse und Prettyprinting ein konkretes (Teil-)System, das einen Parser und einen Prettyprinter für Typausdrücke realisiert, erzeugt werden. Die Referenzarchitektur wird als Sammlung unabhängiger und polymorph verwendbarer Komponenten beschrieben. `makeBasic` realisiert ein Teilsystem bestehend aus einem Modul grundlegender Listenoperationen. `makeLexical` ist ein Teilsystem zur Erzeugung eines Scanners, der über die Schlüsselwörter einer Sprache parametrisiert ist. `makeParse` ist ein Teilsystem von Parserkombinatoren, mit dem auf einfache Art rekursive Abstiegsparser für verschiedenste Grammatiken definiert werden können. `makePretty` unterstützt die Realisierung von Prettyprintern unabhängig von einer konkreten Sprache.

Neben diesen universell einsetzbaren Teilsystemen besteht die Referenzarchitektur aus den spe-

```
(* Eine Basisbibliothek *)
interface BASIC =
  sig
    exception Lookup
    ...
    val minl : int list -> int
    val maxl : int list -> int
    ...
  end

module makeBasic =  define Basic : BASIC
                <- Basic
```

```
(* Toolkit: Lexikalische Analyse *)
interface KEYWORD =
  sig
    val alphas  : string list
    val symbols : string list
  end

interface LEXICAL =
  sig
    datatype token = Id of string
                   | Key of string
    val scan : string -> token list
  end

module makeLexical =
   connect Basic   : BASIC
-> connect Keyword : KEYWORD
-> define  Lexical : LEXICAL
<- Lexical
```

```
(* Toolkit: Parserkombinatoren *)
interface PARSE =
  sig
    exception SynError of string
    type token
    val reader : (token list -> ...
    ...
  end

interface PARSE_TOKEN =
   all lexerToken : type
=> (PARSE with sig
                 type token = lexerToken
               end )
```

```
module makeParse =
   connect Lex   : LEXICAL
-> define Parse : PARSE_TOKEN(Lex.token)
<- Parse
```

```
(* Toolkit: Pretty Printer *)
interface PRETTY =
  sig
    type T
    val blo : int * T list -> T
  end

module makePretty =  define Pretty : PRETTY
                 <- Pretty
```

```
(* Lesen und Schreiben von Typausdruecken *)
module makeLamKey =
   define LamKey : KEYWORD is
     struct
       val alphas = []
       val symbols = ["(", ")", "'", "->"]
     end
<- Lamkey

interface TYPE =
  sig
    datatype typ = Con of string * typ list
                 | Var of string
    val print : typ -> unit
    val read  : string -> typ
  end

module makeLamTypes =
   connect Parse    : PARSE
-> connect Pretty   : PRETTY
-> define  LamTypes : TYPE
<- LamTypes
```

```
(* Zusammenbinden des Subsystems *)
module linkTypesSubSystem =
   bind Basic    to makeBasic
in bind LamKey   to makeLamKey
in bind TypeLex  to makeLexical (Basic)
                                (LamKey)
in bind Parse    to makeParse (TypeLex)
in bind Pretty   to makePretty
in makeLamTypes (Parse)(Pretty)
```

Abbildung 3: Beispielarchitektur

zifischen Teilsystemen `makeLamKey`, das Schlüsselwörter und Symbole für Typausdrücke definiert und `makeLamTypes`, das den Parser und den Prettyprinter für Typausdrücke unter Verwendung der universellen Parserkombinatoren und Prettyprintfunktionen realisiert.

Der Modulausdruck `linkTypesSubSystem` setzt schließlich die verschiedenen unabhängigen Komponenten der Referenzarchitektur für das spezifische Teilsystem zusammen. Durch Ausführung dieses Ausdrucks wird die konkrete Architektur des Teilsystems erzeugt (Abb. 4). Sind alle Im-

```
           define Basic : BASIC
         <- define LamKey : KEYWORD               (* Schluesselwoerter in Typausdruecken *)
      <- define Lexical : LEXICAL                 (* Scanner fuer Typausdruecke *)
   <- define Parse : PARSE_TOKENS (Lexical.token) (* Parserkombinatoren fuer Typausdruecke *)
   <- define Pretty : PRETTY
<- define LamTypes : TYPE                         (* Parser und Printer fuer Typausdruecke *)
<- LamTypes
```

Abbildung 4: Ergebnis der Ausführung von `linkTypesSubSystem`

plementierungen realisiert, so erhält man automatisch die Implementierung des Teilsystems. Im Beispiel ist dies nicht möglich, da lediglich die Implementierung des Moduls `LamKey` angegeben ist.

4 Softwaretechnische Aspekte

Obwohl die vorgestellte Modulsprache viele Ähnlichkeiten zur Modulsprache von Standard ML aufweist – und davon auch inspiriert ist –, werden einige wesentliche softwaretechnische Aspekte besser unterstützt.

Architektur- und Implementierungsbelange sind streng getrennt: Es ist möglich, Referenzarchitekturen als eine Sammlung unabhängiger Modulausdrücke zu entwerfen, ihre Typisierung zu überprüfen und spezifische Architekturen durch Ausführung von Modulprogrammen zu erzeugen, ohne Implementierungen für elementare Module anzugeben. Dies erlaubt das Testen von Referenzarchitekturen und die Generierung verschiedener spezifischer Architekturen.

Typisierung und Seiteneffektfreiheit liefern eine starke Aussage über die statische Korrektheit von Architekturen und ermöglichen gleichungsbasiertes Schließen – Vorteile, die sich direkt aus den Erfahrungen mit typisierten, funktionalen Programmiersprachen übertragen lassen.

Einige eher technische Unterschiede zu SML liefern ebenfalls softwaretechnische Vorteile: So ist es möglich (Implementierungs-)Typen und (Sub-)Module eines Moduls sowohl opak als auch transparent zu spezifizieren; durch die Zuordnung von Implementierungen zu δ-gebundenen (`define`) Namen ist die seperate Übersetzung von Implementierungen möglich.

5 Ausblick

Diese Zusammenfassung kann nur einen kurzen Projektüberblick vermitteln. Der Kern der Arbeit ist das Typsystem der Modulsprache, auf das wir hier nicht näher eingegangen sind. Der nächste Schritt ist eine Implementierung der Modulsprache mit Kern-SML als Implementierungssprache für Module. Darauf aufbauend ist Toolunterstützung unbedingt notwendig, um Schnittstellen- und Modulausdrücke in geeigneter Form (nicht nur textuell) inspizieren und analysieren zu können.

Der Kalkül und die Modulsprache werden im Rahmen des laufenden DFG-Projekts „Semantikbasiertes Entwerfen von Software-Architekturen“ (Förderkennzeichen Sn 11/4-1) entwickelt.

[1] R. Burstall. Programming with modules as typed functional programming. In *Int. Conf. on 5th Generation Computing Systems*. Tokyo, 1984.

[2] D. MacQueen, M. Tofte. A semantics for higher-order functors. *ESOP '94*, LNCS 788, 1994.

[3] X. Leroy. Applicative functors and fully transparent higher-order modules. *22th POPL*, 1995.

[4] D. MacQueen. Using dependent types to express modular structure. *13th POPL*, 1986.

[5] H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science, Vol. 2*, S. 117-309. Oxford Science Publications, 1992.

[6] J. Mitchell, G. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3), 1988.

[7] L. C. Paulson. ML for the Working Programmer. Cambridge University Press, 1991.