# M – eine typisierte, funktionale Sprache für das Programmieren-im-Großen

Kurzfassung

Franz-Josef Grosch *

TU Braunschweig, Abt. Softwaretechnologie
grosch@ips.cs.tu-bs.de

**Zusammenfassung.** Programmieren-im-Großen (PG) kann man als typisiertes, funktionales Programmieren verstehen [3]. Um diese Denkweise programmiersprachlich zu unterstützen, haben wir M, eine typisierte, funktionale Sprache für das PG, entwickelt. In M bilden Module die elementaren Werte und Schnittstellen die elementaren Typen; funktionale Ausdrücke beschreiben die Komponenten von Familien von Software-Systemen. Durch Reduktion der Ausdrücke werden Komponenten zu einzelnen Software-Systemen kombiniert und gebunden.

Grundlage von M ist ein erweiterter $\lambda$-Kalkül, der die Instanziierung von Modulen syntaktisch explizit macht. Schnittstellen sind als *dependent types* [9] formalisiert, und die operationale Semantik ist durch Reduktionsregeln definiert. Diese Arbeit skizziert M und in der gebotenen Kürze den zugrundeliegenden Modulkalkül.

## 1 Einleitung

*Programmieren-im-Großen – typisiertes, funktionales Programmieren?*

Die elementaren Bausteine des Programmierens-im-Großen (PG) sind Module. Jedes Modul besitzt eine *Schnittstelle* und eine *Implementierung*. Nach Parnas [11] charakterisiert die Schnittstelle *alle* Bezeichner, die von einem Modul exportiert werden – alles andere gehört zur Implementierung des Moduls. Schnittstellen und Implementierungen können andere Module zur Benutzung *importieren*. Betrachtet man Module als die elementaren Werte, so bilden Schnittstellen die elementaren Typen des PG.

Typisiertes, funktionales PG bedeutet, die Konstruktion großer Software-Systeme als ein typkorrektes, funktionales Programm zu beschreiben. Jede *Komponente* der Architektur eines Software-Systems wird als funktionaler Ausdruck repräsentiert. Durch Reduktion dieser Ausdrücke werden die zugehörigen Module zum gewünschten Software-System montiert.

Ein *System* ist eine geordnete Folge von Modulen, die zusammen eine Komponente der Architektur bilden. Üblicherweise sind die Module eines Systems entsprechend ihrer Import-Abhängigkeiten topologisch geordnet, um separate

* Die lange Version dieses Artikels finden Sie als Informatik-Forschungsbericht unter http://www.cs.tu-bs.de/softech. Die DFG fördert diese Arbeit unter Kennziffer Sn11/4-1. Mein besonderer Dank gilt Andreas Rossberg.

Übersetzung der einzelnen Modul-Implementierungen zu ermöglichen. Betrachten wir Module als die elementaren Werte, so sind Systeme zusammengesetzte Werte (Aggregate) des PG.

Um ein importiertes Modul typkorrekt zu benutzen, reicht es aus, seine Schnittstelle zu kennen. In vielen Fällen ist es daher sinnvoll, von importierten Modulen zu abstrahieren und Komponenten zu *parametrisieren*. So können einzelne Komponenten unabhängig voneinander entwickelt werden. Ein Software-System wird dann konstruiert, indem schichtweise parametrisierte Komponenten auf bereits zur Verfügung stehende Komponenten appliziert werden. Parametrisierte Komponenten sind die *Funktionen* des PG.

Systeme und Funktionen können, analog zu den Ausdrücken eines funktionalen Programms, als Teil-Systeme oder Hilfs-Funktionen einer größeren Komponente verwendet werden. Funktionales PG erlaubt daher nicht nur die Beschreibung der Architektur eines einzelnen Software-Systems. Vielmehr kann eine Folge benannter Komponenten die Architektur einer Familie von Software-Systemen [12] beschreiben. Die Konstruktion eines bestimmten Software-Systems erfolgt dann durch Auswertung des entsprechenden Ausdrucks.

*Sprachunterstützung für typisiertes, funktionales PG*

Schnittstellen als Typen von Modulen findet man in den meisten Modulsystemen typisierter Programmiersprachen; Modula-2 ist dafür ein Prototyp. Schnittstellen erlauben dem Compiler, die Konsistenz der Modul-Implementierung zu überprüfen; sie ermöglichen separate Übersetzung von Modulen ohne die Implementierungen importierter Module berücksichtigen zu müssen.

Mehrere Implementierungen zu einer Schnittstelle sind eine natürliche Erweiterung des Konzepts Schnittstellen als Typen; wie z. B. in Modula-3 . Schnittstellen importieren jedoch selbst Module; sie sind also Typen, die von Werten abhängen – sie sind *value-dependent types* [9, 1]. Umgekehrt kann eine Implementierung durch verschiedene Schnittstellen charakterisiert werden; dazu muß es eine allgemeinste Schnittstelle, einen *principal type* geben, der eine Sub-Schnittstelle aller passenden Schnittstellen ist – Schnittstellen stehen in einer *subtype*-Relation.

Parametrisierte Module findet man beispielsweise in Ada (*generic packages*) und in der Modulsprache von Standard ML (SML) [10][2]. Parametrisierte Module werden durch Applikation instanziiert. Dabei entsteht das Problem, zu definieren, ob zwei gleiche Applikationen die gleiche Instanz oder zwei verschiedene Instanzen erzeugen. Beispiel: Enthält ein parametrisiertes Modul die Definition eines abstrakten Typs, so definiert jede Instanz einen neuen Typ, der inkompatibel zu allen anderen Typen ist.

In SML erzeugt jede Applikation eines parametrisierten Moduls eine neue Instanz (*structure generativity*). Die operationale Semantik ist *imperativ*, da die Module eines Software-Systems Schritt für Schritt durch Applikation parametrisierter Module auf bereits erzeugte Komponenten instanziiert werden. Andere

[2] In Standard ML heißt ein Modul *structure*, ein parametrisiertes Modul heißt *functor*; die Applikation heißt folglich *functor application*.

Varianten von ML, wie z. B. das Modulsystem von Objective Caml [8], erzeugen ebenfalls mit jeder Applikation neue Instanzen, während die zum Modul gehörigen abstrakten Typen für gleiche Applikationen kompatibel sind – eine fragwürdige Semantik, bedenkt man, daß andere generative Elemente, wie beispielsweise modullokaler Speicher, von Instanz zu Instanz verschieden sind.

Diese Arbeit stellt M vor. M folgt konsequent der Idee des typisierten, funktionalen PG. Das bedeutet:

- Module sind die elementaren Werte
- Schnittstellen sind die elementaren Typen
- Komponenten (Systeme und Funktionen) werden durch Ausdrücke in M repräsentiert
- Alle Ausdrücke sind typisiert
- Auswertung von M-Ausdrücken ist durch eine funktionale Reduktionssemantik definiert
- Reduktion von typkorrekten M-Ausdrücken ist streng normalisierend

Die formale Grundlage für M ist ein typisierter Modulkalkül. Der Modulkalkül ist im wesentlichen ein erweiterter $\lambda$-Kalkül. Das Typsystem des Modulkalküls basiert auf *value-dependent types*, die Schnittstellen von Modulen und zusammengesetzten Ausdrücken erklären, mit einer *subtyping*-Relation zwischen Schnittstellen. Reduktionsregeln definieren die operationale Semantik und lösen auf einfache Weise das Problem der Identifikation der Instanzen von Modulen.

Im folgenden Abschnitt stellen wir die Konzepte von M anhand von Beispielen vor. Den Modulkalkül skizzieren wir kurz im Abschnitt 3. Abschließend versuchen wir einen Vergleich mit der Modulsprache von SML, die sich in verschiedenen formal-technischen und programmier-technischen Aspekten von M unterscheidet.

## 2 Typisiertes, funktionales PG mit M

Darstellbare Beispiele für das PG sind oft klein; möglicherweise zu klein, um die Komplexität des PG zu erfassen. Wir benutzen im folgenden sehr einfache Beispiele aus dem Bereich abstrakter Datentypen, in der Hoffnung, die Konzepte von M zu verdeutlichen.

M als Sprache für das PG ist prinzipiell für jede typisierte Programmiersprache geeignet. Die zugrundeliegende Programmiersprache, die wir *Implementierungssprache* nennen, dient zur Implementierung von Modulen und bestimmt, wie Modulschnittstellen in M definiert werden. Im folgenden legen wir eine Implementierungssprache zugrunde, die an die Kernsprache von SML (*Core SML*) angelehnt ist und intuitiv verständlich sein sollte.

Modul-Schnittstellen spezifizieren die Typen (und die *Kinds*) [4] der Bezeichner, die von einem Modul zur Verfügung gestellt werden. Typische Schnittstellen charakterisieren beispielsweise: einen oder mehrere abstrakte Datentypen mit zugehörigen Operationen, eine Kollektion zusammengehöriger Routinen oder eine Sammlung von Typdeklarationen.

Selbst oft sehr komplex, bilden die Modul-Schnittstellen die elementaren Typen von M. Das Schlüsselwort `interface` kennzeichnet eine Schnittstellendeklaration.

```
interface INTEGER_ORDER = sig
                            type t = int
                            datatype order = LESS | EQUAL | GREATER
                            val compare: t * t -> order
                          end
```

Die Modul-Schnittstelle `INTEGER_ORDER`: ein Typsynonym `t` für den vordefinierten Typ `int`, einen Aufzählungstyp `order`, sowie den Typ einer Funktion `compare`, die die Ordnung zweier Werte vom Typ `t` bestimmt. Eine allgemeinere Schnittstelle `ORDER` definiert den Typ `t` abstrakt (opak).

```
interface ORDER = sig
                    type t
                    datatype order = LESS | EQUAL | GREATER
                    val compare: t * t -> order
                  end
interface INTEGER_ORDER = ORDER with sig type t = int end
```

`INTEGER_ORDER` läßt sich nun als Sub-Schnittstelle von `ORDER` definieren. Dies geschieht durch `with`, indem zusätzliche Einträge zu einer Schnittstelle hinzugefügt und abstrakte Typen durch Konkretisierung verfeinert werden.

Jede Komponente der Architektur eines Software-Systems ist ein benannter M-Ausdruck, eingeleitet durch das Schlüsselwort `component`. Die denkbar einfachste Komponente ist ein vollständiges System bestehend aus einem einzelnen Modul. `JustIntOrder` ist dafür ein Beispiel:

```
component JustIntOrder = dec[IntegerOrder: INTEGER_ORDER]
                         export IntegerOrder
```

`JustIntOrder` deklariert (`dec`) ein Modul `IntegerOrder`, das durch die Schnittstelle `INTEGER_ORDER` charakterisiert wird. `dec` ist ein Bindungskonstrukt, das den Modulnamen `IntegerOrder` im nachfolgenden Rumpf des Deklarationsausdrucks bindet. Im allgemeinen ist der Rumpf ein beliebiger M-Ausdruck. In diesem Fall wird im Rumpf lediglich definiert, daß das Modul `IntegerOrder` exportiert wird. Betrachten wir `JustIntOrder` als ein vollständiges Software-System, so bildet `IntegerOrder` das Hauptmodul. Für die Wiederverwendung als Teil-System definiert der Exportteil, welche Module von außen benutzt werden können.

Fügen wir einen Modulparameter (`fun`) zu einem System hinzu, so erhalten wir eine parametrisierte Komponente – eine Funktion, die, angewandt auf ein passendes Modul, ein System berechnet.

```
component MakeDictionary = fun (Key: ORDER)
                           dec [Dictionary: DICTIONARY Key]
                           export Dictionary
```

Die Komponente `MakeDictionary` ist parametrisiert mit einem Modul, das zur Schnittstelle `ORDER` paßt. Das Modul `Dictionary` benutzt eine Ordnung als Suchschlüssel. Dies wird auch in der Schnittstelle `DICTIONARY` deutlich.

```
interface DICTIONARY (Key: ORDER) =
  sig
    type key = Key.t
    type entry
    type dictionary
    val empty: dictionary
    val lookup: dictionary * key -> entry
    val insert: dictionary * key * entry -> dictionary
    ...
  end
```

Die Schnittstelle `DICTIONARY` ist parametrisiert über ein Ordnungsmodul. Dies ist notwendig, da verschiedene Ordnungen zu inkompatiblen Typen `key` führen. Parametrisierte Schnittstellen sind die parametrisierten Typen von M. In der Funktionen `MakeDictionary` wird die parametrisierte Schnittstellen mit dem Modul-Parameter `Key` instanziiert.

Im nächsten Schritt applizieren wir `MakeDictionary` auf ein Ordnungsmodul. Zu diesem Zweck benutzen wir das System `JustIntOrder` als Teil-System und kombinieren es mit der Funktion – Montage von vorgefertigten Komponenten. Da `JustIntOrder` kein Modul sondern ein vollständiges System ist, das ein passendes Modul exportiert, müssen wir dieses System zunächst öffnen (`use`), um auf den Exportteil zuzugreifen. Dann kann die Funktion auf das exportierte Modul appliziert werden. Der M-Ausdruck, der die Komponenten kombiniert, hat folgende Form:

```
component IntKeyDictionary = use JustIntOrder            as Key
                             in  MakeDictionary Key
```

`IntKeyDictionary` beschreibt die Architektur eines vollständigen Software-Systems, das durch Reduktion berechnet wird. Wir werden sehen, daß die Systeme `JustIntOrder` und `IntKeyDictionary` unabhängig voneinander sind und jeweils alle Module umfassen, aus denen die Systeme zusammengesetzt sind. Die Vorstellung, beide Systeme referenzierten das Modul `IntegerOrder`, ist falsch. Richtig ist, daß je eine Instanz des Moduls Teil jedes Systems ist - die Systeme sind funktionale Ausdrücke.

Um die Berechnung zu verstehen, betrachten wir die Reduktionsregeln für `use` und die Funktionsapplikation.

```
use (dec[a:A] export m) as x in n   ⟶   dec[a:A] (n{x ← m})
(fun(x:A) export m) n               ⟶   m{x ← n}
```

Der `use`-Ausdruck kombiniert das wiederverwendete System `dec[a:A] m` mit dem Ausdruck `n`. Dabei wird der Exportteil des Systems an die Variable `x` gebunden, die im Ausdruck `n` sichtbar ist[3]. Das Ergebnis der Reduktion ist ein vollständiges System, das sowohl die Module des wiederverwendeten Systems als auch den Ausdruck `n`, in dem die Variable `x` durch den Exportteil des wiederverwendeten Systems ersetzt ist, enthält.

[3] Die Reduktionsregel für `use` ist hier vereinfacht dargestellt. Siehe auch Abschnitt 3.

Die Applikation einer Funktion auf ein passendes Argument ist eine einfache Funktionsapplikation und wird durch $\beta$-Reduktion berechnet. Selbstverständlich ist in beiden Fällen die hier nicht dargestellte $\alpha$-Konversion zu berücksichtigen, um irrtümliche Bindungen von Bezeichnern zu verhindern.

Wendet man die Reduktionsregeln auf den Ausdruck `IntKeyDictionary` an, so erhält man folgende Reduktionsschritte.

```
use JustIntOrder as Key in MakeDictionary Key
≡ use (dec[IntegerOrder: INTEGER_ORDER] export IntegerOrder) as Key
  in MakeDictionary Key
≡ dec[IntegerOrder: INTEGER_ORDER]
  (MakeDictionary Key){Key ← IntegerOrder}
≡ dec[IntegerOrder: INTEGER_ORDER] (MakeDictionary IntegerOrder)
≡ dec[IntegerOrder: INTEGER_ORDER]
  ((fun(Key: ORDER) dec[Dictionary: DICTIONARY Key] export Dictionary)
  IntegerOrder)
≡ dec[IntegerOrder: INTEGER_ORDER]
  dec[Dictionary: DICTIONARY IntegerOrder]
  export Dictionary
```

Das Ergebnis der Reduktion von `IntKeyDictionary` ist ein vollständiges, geschlossenes System, repräsentiert als M-Ausdruck. Es besteht aus den beiden Modulen `IntegerOrder` und `Dictionary` und exportiert Modul `Dictionary` als Hauptmodul oder als Schnittstelle zur Wiederverwendung von außen.

Benötigen wir ein System, das das Ordnungsmodul, das Verzeichnis und zusätzlich etwa eine Warteschlange kombiniert, wobei sowohl das Verzeichnis als auch die Warteschlange dasselbe Ordnungsmodul referenzieren, so schreiben wir:

```
component QueueAndDict =
   use JustIntOrder             as order       in
   use MakePriorityQueue order  as queue       in
   use MakeDictionary order     as dictionary  in
   {Order=order, Queue=queue, Dictionary=dictionary}
```

Auch diese Komponente ist vollständig, ohne offene Referenzen. Die Komponenten `JustIntOrder`, `MakeDictionary` und die nicht näher spezifizierte Komponente `MakePriorityQueue` werden als Teil-Systeme und Hilfs-Funktionen wiederverwendet, um daraus das gewünschte System zusammenzusetzen. `QueueAndDict` exportiert alle drei enthaltenen Module in einem Tupel. Tupelbildung und -selektion ist immer dann notwendig, wenn der Exportteil eines Systems aus mehr als einem Modul bestehen soll.

### 2.1 Modul-Implementierungen

Die M-Ausdrücke im vorangehenden Abschnitt beschreiben die modulare Struktur mehrerer, teilweise parametrisierter Komponenten. Die elementaren Bausteine dieser Komponenten sind die Module, die durch Angabe eines Namens und einer Schnittstelle deklariert (`dec`) werden. Alle angegebenen Komponenten sind

konstruiert aus den drei elementaren Modulen `IntegerOrder`, `Dictionary` und `PriorityQueue`. Modul-Implementierungen fehlen bisher.

Mit der zusätzlichen Einschränkung, daß die Bezeichner für Parameter (`fun`) und Deklaration (`dec`) disjunkt sind, können Implementierungen wie folgt hinzugefügt werden.

```
implementation IntegerOrder of JustIntOrder =
  struct
    type t = int
    datatype order = LESS | EQUAL | GREATER
    fun compare (i, j) =  ...
  end
implementation Dictionary of MakeDictionary =
  struct  import Key
    type key = Key.t                      (* Key is an ordering *)
    type entry = string                   (* entries are strings *)
    type dictionary = (key * entry) list (* association list *)
    val empty = []                        (* empty list *)
    fun lookup (dict, key) = ...
    fun insert (dict, key, entry) = ...
  end
```

Die Implementierung eines Moduls erfolgt im selben Sichtbarkeitsbereich, wie die Deklaration des Moduls. Betrachten wir Implementierungen und Schnittstellen genauer, so erkennen wir, daß beide von Modulen im Sichtbarkeitsbereich abhängen. Während der Reduktion von M-Ausdrücken, werden Substitutionen nicht nur in Ausdrücken und Schnittstellen, sondern auch in den zugehörigen Implementierungen vorgenommen. Letztere etablieren die Bindungen der Modul-Implementierungen.

Eine Komponente ohne Parameter, ein System, repräsentiert immer auch ein vollständiges Software-System. Sind alle Modul-Implementierungen angegeben, so kann ein System zu einem ausführbaren Programm gebunden werden.

### 2.2 Schnittstellen zusammengesetzter Ausdrücke

M ist eine typisierte, funktionale Sprache. Daher werden nicht nur Module durch eine Schnittstelle charakterisiert, sondern jeder Ausdruck ist typisiert – er hat eine zusammengesetzte Schnittstelle. Die Typüberprüfung für M-Ausdrücke berechnet für die Komponente `JustIntOrder` eine zusammengesetzte Schnittstelle, die zu folgender Schnittstellendeklaration äquivalent ist:

```
interface JUST_INT_ORDER = some[IntegerOrder: INTEGER_ORDER]
                           export INTEGER_ORDER
```

`JUST_INT_ORDER` ist ein existenz-quantifizierter *dependent type*, eine *weak dependent sum* [14]. Dies lesen wir: Es gibt ein Modul `IntegerOrder` mit Schnittstelle `INTEGER_ORDER` als Teil eines Systems, das ein Modul mit Schnittstelle `INTEGER_ORDER` exportiert. Die parametrisierte Komponente `MakeDictionary` wird durch folgende zusammengesetzte Schnittstelle charakterisiert:

```
interface MAKE_DICTIONARY = all(Key: ORDER)
                            some[Dictionary: DICTIONARY Key]
                            export (DICTIONARY Key)
```

Die Schnittstelle `MAKE_DICTIONARY` ist die Kombination eines *dependent function type* mit einem *weak dependent sum type*: Für alle Module `Key` mit Schnittstelle `ORDER` gibt es ein Modul `Dictionary` mit der von Modul `Key` abhängigen Schnittstelle `DICTIONARY Key`, so daß ein Modul mit Schnittstelle `DICTIONARY Key` exportiert wird.

## 3 Der Modulkalkül – formale Grundlage von M

Die Grundlage von M bildet ein Modulkalkül, der im wesentlichen ein erweiterter, typisierter $\lambda$-Kalkül ist. Die zentrale Erweiterung ist die Deklaration von Modulen (`dec`) und die zugehörige Verwendung (`use`). Charakteristisch für die Typisierung des Modulkalküls sind Typen (Schnittstellen), die von Werten (Ausdrücken) abhängen, *value-dependent types* [1, 14]. Die operationale Semantik des Modulkalküls ist durch Reduktionsregeln definiert. Die Reduktion ist *konfluent* und *streng normalisierend*. Dies bedeutet einerseits, daß die Reduktionsreihenfolge kein Rolle spielt, andererseits terminiert jede Reduktiom mit eindeutiger Normalform.

Abbildung 1 skizziert Syntax, Reduktion und Typisierung des Modulkalküls. Ein ausführlichere Darstellung findet man in [7] und [6].

## 4 Vergleich und Zusammenfassung

Die Modulsprachen der ML-Familie, d. h. SML und dessen Varianten Objective Caml [8] und SML'96 [13], sind die einzigen Programmiersprachen, die vergleichbar mit M das PG unterstützen. Allerdings gibt es große Unterschiede zwischen den Modulsprachen der ML-Familie und M. In M ist das PG von der Implementierung von Modulen strikt getrennt. Dies erlaubt einerseits, M prinzipiell für verschiedene typisierte Implementierungssprachen zu benutzen, andererseits kann die Entwicklung der Architektur eines Software-Systems unabhängig von Modul-Implementierungen erfolgen. In den ML-Varianten sind Implementierungssprache und Modulsprache stark gekoppelt. Ohne Modul-Implementierungen ist keine Auswertung von modulsprachlichen Ausdrücken möglich; parametrisierte Module (*functor*) liefern als Ergebnis immer eine Modul-Implementierung (*structure*).

In ML-Programmen folgen Ausdrücke der Modulsprache und Ausdrücke der Kernsprache in beliebiger Reihenfolge. Folglich hängt die Auswertung von modulsprachlichen Ausdrücken vom Zustand ab. Dies führt zu einer dynamischen Semantik, die Instanzen Schritt für Schritt, gewissermaßen imperativ, erzeugt. In M ist das Ergebnis der Auswertung eines Ausdrucks wiederum ein Ausdruck, der die modulare Struktur einer Komponente repräsentiert. Durch mehrere Komponenten lassen sich so, unabhängig voneinander, mehrere Software-Systeme (z. B. eine Familie von Software-Systemen) darstellen. In ML repräsentieren Module die Komponenten einer Architektur, während in M komplexe Ausdrücke die Komponenten bilden.

Die Trennung des PG von der Implementierung der Module erlaubt die prototypische Entwicklung von Software-Architekturen [5], ohne die Notwendigkeit,

Syntax

| | | | |
|---|---|---|---|
| Ausdrücke | $m$ | $::= x$ | Variable/Name |
| | | $\mid l$ | Label |
| | | $\mid \mathsf{fun}(x:I)\ m$ | (Funktions-)Abstraktion |
| | | $\mid m_1\ m_2$ | Applikation |
| | | $\mid \langle l_1 = m_1, \ldots, l_n = m_n \rangle$ | Tupelkonstruktion |
| | | $\mid m\#l$ | Tupelselektion |
| | | $\mid \mathsf{dec}[x:S]\ m$ | (System-)Deklaration |
| | | $\mid \mathsf{use}\ m_1\ \mathsf{as}\ [x_1]x_2\ \mathsf{in}\ m_2$ | Verwendung |
| Schnittstellen | $I$ | $::= S$ | Modul-Schnittstelle |
| | | $\mid \mathsf{all}(x:I_1)\ I_2$ | Funktions-Schnittstelle |
| | | $\mid \langle l_1 : I_1, \ldots, l_1 : I_n \rangle$ | Tupel-Schnittstelle |
| | | $\mid \mathsf{some}[x:S]\ I$ | System-Schnittstelle |
| Signaturen | $S$ | $::= \mathsf{sig}\ E\ \mathsf{end}$ | Signatur |
| | $E$ | $::= D, E$ | Signatureinträge |
| | | $\mid \varepsilon$ | |
| | $D$ | $::= \mathsf{type}\ t$ | opake Typspezifikation |
| | | $\mid \mathsf{type}\ t = T$ | manifeste Typspezifikation |
| | | $\mid \mathsf{val}\ v : T$ | Wertspezifikation |
| | $T$ | $::= t$ | Typname |
| | | $\mid m.t$ | qualifizierter Typ |
| | | $\mid \mathit{int} \mid \mathit{bool}$ | vordefinierte einfache Typen |
| | | $\mid T_1 \to T_2$ | Funktionstyp |

Reduktionsregeln

$$(\mathsf{fun}(x:A)\ m_1)\ m_2 \longrightarrow m_1\{x \leftarrow m_2\}$$

$$\langle l_1 = m_1, \ldots, l_k = m_k, \ldots, l_n = m_n \rangle \# l_k \longrightarrow m_k\{l_{k-1} \leftarrow m_{k-1}\} \cdots \{l_1 \leftarrow m_1\}$$

$$\mathsf{use}\ (\mathsf{dec}[x_1:S]\ m_1)\ \mathsf{as}\ [x_2]x_3\ \mathsf{in}\ m_2 \longrightarrow \mathsf{dec}[x_1:S]\ (m_2\{x_2 \leftarrow x_1\}\{x_3 \leftarrow m_1\})$$

Typisierungsregeln (Auswahl)

(Modul)
$$\frac{\Gamma \vdash m : \mathsf{sig}\ \mathsf{type}\ t, E\ \mathsf{end}}{\Gamma, \mathsf{type}\ t \vdash m : \mathsf{sig}\ E\ \mathsf{end}}$$

(Abstraktion)
$$\frac{\Gamma, x:I_1 \vdash m : I_2 \quad \Gamma \vdash \mathsf{all}(x:I_1)\ I_2 : \square}{\Gamma \vdash \mathsf{fun}(x:I_1)\ m : \mathsf{all}(x:I_1)\ I_2}$$

(Applikation)
$$\frac{\Gamma \vdash m_1 : \mathsf{all}(x:I_2)\ I_1 \quad \Gamma \vdash m_2 : I_2}{\Gamma \vdash m_1\ m_2 : I_1\{x \leftarrow m_2\}}$$

(Tupelkonstruktion)
$$\frac{\Gamma \vdash \langle l_2 = m_2, \ldots \rangle\{l_1 \leftarrow m_1\} : \langle l_2 : I_2, \ldots \rangle\{l_1 \leftarrow m_1\} \quad \Gamma \vdash m_1 : I_1 \quad \Gamma \vdash \langle l_1 : I_1, l_2 : I_2, \ldots \rangle : \square}{\Gamma \vdash \langle l_1 = m_1, l_2 = m_2, \ldots \rangle : \langle l_1 : I_1, l_2 : I_2, \ldots \rangle}$$

(Tupelselektion)
$$\frac{\Gamma \vdash m : \langle l_1 : I_1, \ldots, l_k : I_k, \ldots, l_n : I_n \rangle}{\Gamma \vdash m\#l_k : I_k\{l_{k-1} \leftarrow m\#l_{k-1}\} \cdots \{l_1 \leftarrow m\#l_1\}}$$

(Deklaration)
$$\frac{\Gamma, x:I_1 \vdash m : I \quad \Gamma \vdash \mathsf{some}[x:S]\ I : \square}{\Gamma \vdash \mathsf{dec}[x:S]\ m : \mathsf{some}[x:S]\ I}$$

(Verwendung)
$$\frac{\Gamma \vdash m_1 : \mathsf{some}[x_1:S]\ I_1 \quad \Gamma, x_2:S, x_3:I_1 \vdash m_2 : I_2}{\Gamma \vdash \mathsf{use}\ m_1\ \mathsf{as}\ [x_2]x_3\ \mathsf{in}\ m_2 : \mathsf{some}[x_1:S]\ (I_2\{x_2 \leftarrow x_1\})}, x_3 \notin I_2$$

**Abbildung1.** Syntax, Reduktion und Typisierung des Modulkalküls

Implementierungen angeben zu müssen. Insbesondere Referenzarchitekturen [15] lassen sich mit Hilfe von M-Ausdrücken definieren und durch Reduktion instanziieren.

Die Motivation für M entstand aus der Suche nach programmiersprachlicher Unterstützung bei der Entwicklung von Software-Architekturen, insbesondere Referenzarchitekturen. Es sollte möglich sein, Software-Systeme im Sinne einer Komponententechnologie [2] zu entwerfen und zusammenzubauen, ohne auf die wertvolle Unterstützung moderner Typisierungstechniken und zustandsfreier Auswertung zu verzichten.

In einem ersten Prototyp haben wir M für den Kern von SML als Implementierungssprache entwickelt. Zwei wichtige Projektrichtungen eröffnen sich für die Zukunft: die Entwicklung einer leistungsfähigen, bereichsspezifischen Referenzarchitektur in M zur Demonstration der Vorteile beim PG und die Anwendung des Modulkalküls auf eine typisierte objekt-orientierte Sprache.

## Literatur


1. Barendregt, H. P.: Lambda calculi with types. In *Handbook of Logic in Computer Science*, Band 2, S. 117–309. Oxford University Press, New York, 1992.
2. Batory, D. und O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, 1(4):355–398, Okt. 1992.
3. Burstall, R. M.: Programming with modules as typed functional programming. In *Proc. Internation Conference on 5th Generation Computing Systems*, Tokyo, 1984.
4. Cardelli, L.: Typeful programming. In *Formal Description of Programming Concepts*, S. 431–507. Springer Verlag, 1991.
5. Garlan, D. und Perry, D. E.: Introduction to the special issue on software architecture. *IEEE Trans. on Software Engineering*, 21(4):269–274, April 1994.
6. Grosch, F. J.: A syntactic approach to structure generativity. Informatik-Bericht 96-05, TU Braunschweig, Juli 1996.
7. Grosch, F. J.: M - eine typisierte, funktionale Sprache für das Programmieren-im-Großen. Informatik-Bericht, TU Braunschweig, Mai 1997.
8. Leroy, X.: Le système Caml Special Light: modules et compilation efficace en Caml. Bericht 2721, INRIA, November 1995. Auf Französisch.
9. MacQueen, D.: Using dependent types to express modular structure. In *13th POPL*, S. 277–286. ACM Press, Januar 1986.
10. Milner, R., Tofte, M. und Harper, R.: *The Definition of Standard ML*. 1990.
11. Parnas, D. L.: Designing software for ease of expansion and contraction. *IEEE Trans. on Software Engineering*, SE-5(2):128–138, März 1979.
12. Parnas, D. L.: On the design and development of program families. *IEEE Trans. on Software Engineering*, SE-2(1):1–9, März 1976.
13. Stone, C. und Harper, R.: A type-theoretic account of Standard ML 1996. Bericht CMU-CS-96-136, School of Computer Science, CMU, Mai 1996.
14. Thompson, S.: *Type Theory and Functional Programming*. Addison-Wesley, 1991.
15. Tracz, W.: DSSA (Domain-specific software architecture) pedagogical example. *ACM SIGSOFT Software Engineering Notes*, 20(3):49–62, Juli 1995.


Dieser Artikel wurde mit dem LaTeX Makro-Paket und dem LLNCS-Style formatiert.