Erweiterbare Mini-Sprache Core

Uwe Kastens, Universität-GH Paderborn

1. Dezember 1999

Inhaltsverzeichnis

1 Übersicht

Die Mini-Sprache Core ist eine sehr einfache imperative, statisch typisierte Programmiersprache. Dieses Dokument beschreibt die Herstellung eines Übersetzers, der Core-Programme in C-Programme übersetzt. An diesem Beispiel werden wichtige Methoden und Techniken zur Generierung vollständiger Übersetzer mit Eli demonstiert. Das Dokument ist so geschrieben, daß Eli daraus den Übersetzer generiert. Zum Verständnis der Spezifikationen und der Erläuterungen sollten die entsprechenden Abschnitte der Eli-Dokumentation herangezogen werden.
Dieses Dokument enthält drei verschiedene Arten von Aufgaben, die wie folgt gekennzeichnet sind:
I-Aufgabe: Sie sollen Ihr Verständnis von Spezifikationstechniken prüfen und vertiefen. Zur Lösung benötigen Sie in einigen Fällen die Eli-Dokumentation oder das Eli-System, um Ableitungen auszuführen.
U-Aufgabe: In diesem Dokument ist die Übersetzung nach C noch unvollständig. Diese Aufgaben leiten Sie an, sie vollständig zu spezifizieren.
E-Aufgabe: Es werden einige kleine Spracherweiterungen vorgeschlagen, mit denen die Sprache für einfache Algorithmen brauchbar wird.
Core umfaßt Deklarationen von Variablen und Konstanten, Zuweisungen und Ein- und Ausgabeanweisungen. Die Typen `int` und `real` sind vordefiniert.
Hier ist ein Beispielprogramm: **simple**[1] $\equiv$

```
program
let
  a:    int;
  x, y: real;
  i = 10;
in
  input a;
  x := 3.5;
  y := a;
  output a;
  output x;
  output y;
  output i;
end
```

This macro is attached to a product file.

Selbstverständlich kann man in dieser einfachen Sprache noch keine sinnvollen Programme schreiben: Die Sprache Core kann durch Hinzufügen von Sprachkonstrukten, wie Ablaufstrukturen, Ausdrücke mit Operatoren und weitere Datentypen, erweitert werden.

I-Aufgabe: Wie werden in diesem Dokument die zwischen den Erläuterungen stehenden Spezifikationsdateien identifiziert?

I-Aufgabe: Generieren Sie aus diesem Dokument einen Core-Übersetzer. Lassen Sie ihn das Beispielprogramm übersetzen. Sehen Sie sich das Ergebnis an. (Eine Übersetzung mit dem C-Übersetzer lohnt erst, wenn Sie die U-Aufgaben gelöst haben.)

Im folgenden werden die Sprachstruktur, die Aufgaben der statischen Semantik (Namensanalyse, Typanalyse) und die Übersetzung nach C spezifiziert.

2 Sprachstruktur

Die Notation von Core-Programmen ist an der von Pascal angelehnt. Wir stellen zunächst die Sprachkonstrukte anhand ihrer konkreten Syntax vor, ohne hier ihre Semantik zu beschreiben.

Ein Programm besteht aus einem Block von Deklarationen und Anweisungen:

Program.con[2] $\equiv$

```
Program:        Source.
Source:         'program' Block.
Block:          'let' Declarations 'in' Statements 'end'.
Declarations:   Declaration*.
Statements:     Statement*.
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich über die Notation von Produktionen der konkreten Syntax, insbesondere von EBNF-Konstrukten wie `Statement*`.

Variable werden unter Angabe ihres Typs deklariert. Als Typangaben sind zunächst nur vordefinierte Typbezeichner zugelassen:

VarDecl.con[3] ≡

```
Declaration:    DefIdents ':' TypeDenoter ';'.
DefIdents:      DefIdent // ','.
DefIdent:       Ident.
TypeDenoter:    UseTypeIdent.
UseTypeIdent:   Ident.
```

This macro is attached to a product file.

Die drei letzten Produktionen sind Kettenproduktionen. Die für `DefIdent` und `UseTypeIdent` wurden eingeführt, um Bezeichnerauftreten nach ihrem syntaktischen Kontext zu klassifizieren, soweit dies für Übersetzungsaufgaben wichtig ist. Die Kettenproduktion für `TypeDenoter` wurde eingeführt, damit man andere Formen von Typangaben ergänzen kann.

Konstantendeklarationen führen Namen für ganzzahlige Werte ein:

ConstDecl.con[4] ≡

```
Declaration:    DefIdent '=' IntLiteral ';'.
```

This macro is attached to a product file.

Eine Anweisung kann eine Zuweisung an eine Variable, eine Eingabe- oder eine Ausgabeanweisung sein.

Statement.con[5] ≡

```
Statement:      UseIdent ':=' Expression ';'.
Statement:      'input' UseIdent ';'.
Statement:      'output' Expression ';'.
```

This macro is attached to a product file.

Die folgenden Kettenproduktionen bilden einen Rahmen für eine Ausdrucksgrammatik mit 6 Präzedenzstufen. Es sind nur Operatoren für Multiplikation und Addition definiert. (Die Operatorzeichen werden unten mit Hilfe eines Moduls ergänzt.) Weitere ein- und zweistellige Operatoren können hier eingefügt werden.

Expression.con[6] ≡

```
Expression:     Expr1.
Expr1:          Expr2.
Expr2:          Expr3.
Expr3:          Expr3 AddOpr Expr4.
Expr3:          Expr4.
Expr4:          Expr4 MulOpr Operand.
Expr4:          Expr5.
Expr5:          Operand.
Operand:        '(' Expression ')'.
```

This macro is attached to a product file.

Die Operatorpräzedenzen sind nur für die Strukturanalyse wichtig. Im Strukturbaum, der daraus aufgebaut wird, unterscheiden wir diese Klassifizierung der Grammatiksymbole nicht. Sie werden als äquivalent spezifiziert:

Expression.sym[7] ≡

```
Expression ::= Expr1 Expr2 Expr3 Expr4 Expr5 Operand .
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich im Dokument (Syntax, Mapping) über die Bedeutung solcher Äquivalenzen.

Elementare Operanden sind Variablenbezeichner oder Literale für ganzzahlige oder Gleitpunktwerte:

Operand.con[8] ≡

```
Operand:        UseIdent.
Operand:        IntLiteral.
Operand:        FloatLiteral.
UseIdent:       Ident.
```

This macro is attached to a product file.

Die Bezeichner, Literale und Kommentare werden wie in Pascal notiert:

Mini.gla[9] ≡

```
Ident:          PASCAL_IDENTIFIER
IntLiteral:     PASCAL_INTEGER
FloatLiteral:   PASCAL_REAL
                PASCAL_COMMENT
```

This macro is attached to a product file.

3 Namensanalyse

Bezeichner treten als `DefIdent` in Deklarationen definierend auf, und angewandt als `UseIdent` für Variable und Konstante, sowie als `UseTypeIdent` für Typen.

Der Gültigkeitsbereich einer Definition ist der gesamte `Block`, der die Definition enthält. Der (hier einzige) `Block` ist in das `Program` eingebettet. Die Schachtelung von Blöcken ist in Core nicht definiert, sie kann als Erweiterung ergänzt werden.

Aufgabe der Namensanalyse ist es, jedes bezeichnete Objekt (Variable oder Konstante) durch einen Schlüssel (`Key`) so zu repräsentieren, daß die Definition und die Anwendungen desselben Objekts den gleichen Schlüssel haben. So kann man dann den Schlüsseln Eigenschaften zuordnen, die an den Anwendungen geprüft oder zur Übersetzung verwendet werden.

Alle Aufgaben der semantischen Analyse und der Transformation werden als Berechnungen im Strukturbaum durchgeführt und in der Sprache LIDO spezifiziert. Die grundlegenden Berechnungen zur Namensanalyse übernehmen wir von folgendem Modul aus der Modul-Bibliothek:

ScopeLibUse.specs[10] ≡

```
$/Name/AlgScope.gnrc:inst
```

This macro is attached to a product file.

I-Aufgabe: Der Modul `AlgScope` implementiert Namensanalyse auf der Basis der Algol-Verdeckungsregel. Ein entsprechender Modul `CScope` basiert auf der C-Verdeckungsregel. Begründen Sie, ob die Verwendung von `CScope` die Sprache ändern würde.

I-Aufgabe: Informieren Sie sich im Dokument (Modlib, Name Analysis) über die Verwendung dieser Module zur Namensanalyse.

Terminalsymbole der Grammatik wie zum Beispiel `Ident` haben einen Wert, der das Terminalsymbol repräsentiert. Er wird von der lexikalischen Analyse berechnet. Um den Modul verwenden zu können, müssen wir den Terminalwert als Attribut mit Namen `Sym` an Symbolknoten verfügbar machen, die Bezeichner in ihren unterschiedlichen Auftreten repräsentieren. Die Berechnung dafür ordnen wir einem `CLASS`-Symbol `IdentOcc` zu, um sie für die drei Bezeichnerklassen wiederverwenden zu können:

IdentOccurrence.lido[11] ≡

```
ATTR Sym: int;
CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;

SYMBOL DefIdent     INHERITS IdentOcc END;
SYMBOL UseIdent     INHERITS IdentOcc END;
SYMBOL UseTypeIdent INHERITS IdentOcc END;
```

This macro is attached to a product file.

I-Aufgabe: Schlagen Sie im LIDO-Dokument die Bedeutung des `INHERITS`-Konstrukts nach.

Der `AlgScope`-Modul stellt Berechnungen zur Verfügung, die abstrakten Symbolen zugeordnet sind: `RootScope` für die Wurzel der Grammatik, `RangeScope` für Bereiche, die die Gültigkeit von Definitionen begrenzen, `IdDefScope` für definierende und `IdUseEnv` für angewandte Auftreten von Bezeichnern. Wir ordnen sie den entsprechenden Symbolen unserer Grammatik zu:

Scope.lido[12] ≡

```
SYMBOL Program          INHERITS RootScope END;
SYMBOL Block            INHERITS RangeScope END;
SYMBOL DefIdent         INHERITS IdDefScope END;
SYMBOL UseIdent         INHERITS IdUseEnv END;
SYMBOL UseTypeIdent     INHERITS IdUseEnv END;
```

This macro is attached to a product file.

I-Aufgabe: Schlagen Sie im AlgScope-Dokument die Beschreibungen dieser Symbolrollen nach.

Mit den so aus der Bibliothek bezogenen Berechnungen steht an jedem Bezeichnerauftreten der Schlüssel als Attribut `Key` zur Verfügung.

Wir müssen nun einige Bedingungen für die Korrektheit der Definition und Anwendung von Bezeichnern spezifizieren:

1. Zu jedem angewandten Bezeichner muß es eine gültige Definition geben.

2. In einem `Block` darf es für einen Bezeichner höchstens eine Definition geben.

Die erste Bedingung ist verletzt, wenn die `IdUseEnv`-Berechnung den Wert `NoKey` als `Key`-Attribut liefert. Diese Prüfung binden wir an ein `CLASS`-Symbol `NoKeyMsg` und ordnen dieses den Symbolen für angewandte Bezeichnerauftreten zu:

ScopeDefChk.lido[13] ≡

```
CLASS SYMBOL NoKeyMsg COMPUTE
  IF (EQ (THIS.Key, NoKey),
  message(ERROR, CatStrInd ("identifier is not defined: ", THIS.Sym),
          0, COORDREF));
END;

SYMBOL UseIdent         INHERITS NoKeyMsg END;
SYMBOL UseTypeIdent     INHERITS NoKeyMsg END;
```

This macro is attached to a product file.

I-Aufgabe: Suchen Sie in der Dokumentation des `AlgScope`-Moduls eine Symbolrolle, die Sie statt `NoKeyMsg` verwenden könnten.

Die Funktion `CatStrInd` zur Konkatenation von Meldungen mit Bezeichnernnamen beziehen wir aus dem einem Bibliotheksmodul:

MessageLibUse.specs[14] ≡

```
$/Tech/Strings.specs
```

This macro is attached to a product file.

I-Aufgabe: Schlagen Sie im Modlib-Dokument nach, welche Parameter die vom `Strings`-Modul bereitgestellten Funktionen haben.

Für die Prüfung der zweiten Bedingung verwenden wir den Bibliotheksmodul `Unique`:

UniqueLibUse.specs[15] ≡

```
$/Prop/Unique.gnrc:inst
```

This macro is attached to a product file.

Seine Berechnungen ordnen wir den Grammatiksymbolen für definierende Bezeichnerauftreten und für die Grammatikwurzel zu. Das Ergebnis der Eindeutigkeitsprüfung, das Attribut `Unique`, verwenden wir, um die Entscheidung für eine Fehlermeldung zu treffen:

UniqueChk.lido[16] ≡

```
SYMBOL Program  INHERITS RangeUnique END;
SYMBOL DefIdent INHERITS Unique COMPUTE
  IF (NOT (THIS.Unique),
  message (ERROR, CatStrInd("identifier is multiply defined: ",
                            THIS.Sym),
  0, COORDREF));
END;
```

This macro is attached to a product file.

In unserer Sprache sind Bezeichner für die Typen `int` und `real` vordefiniert.

Vordefinierte Objekte werden als Definitionen in die Umgebung des Programms eingefügt. Dies leisten die Bibliotheksmodule `PreDefine` und `PreDefId`. `PreDefine` wird mit dem Namen des Grammatiksymbols für Bezeichner (`Ident`) instantiiert.

CorePreDefine.specs[17] ≡

```
$/Name/PreDefine.gnrc +referto=Ident :inst
$/Name/PreDefId.gnrc  +referto=(CorePreDefId.d) :inst
```

This macro is attached to a product file.

Der Modul `PreDefId` wird mit der Eli-Ableitung für die folgende Datei instantiiert. Sie enthält Makro-Aufrufe, welche die vordefinierten Bezeichner `Key`-Konstante zuordnen:

CorePreDefId.d[18] ≡

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
```

This macro is attached to a product file.

I-Aufgabe: Schlagen Sie im Modlib-Dokument nach, wie die Module `PreDefine` und `PreDefId` verwendet werden. Warum geben wir hier den `instance`-Parameter nicht an?

Eigenschaften der vordefinierten Objekte werden im folgenden Abschnitt "Typanalyse" zugeordnet.

E-Aufgabe: Erweitern Sie Core um geschachtelte Blöcke. Müssen Sie dafür etwas an der Namensanalyse ändern?

4 Typanalyse

Die zentrale Aufgabe der Typanalyse ist es, Eigenschaften von Programmobjekten und Programmkonstrukten zu bestimmen und zu prüfen, ob sie gemäß der Sprachregeln korrekt verwendet werden. Wir charakterisieren Programmobjekte und Sprachkontrukte durch ihre Art (Konstante, Variable, Typ) und ihren Typ.

Wir führen dazu die Eigenschaften `Type` und `Kind` ein. Sie werden in Deklarationen den Objekten zugeordnet. An Symbolen für Programmkonstrukte werden sie als Attribute berechnet und überprüft, daß sie korrekt verwendet werden.

Zur Spezifikation der Typanalyse verwenden wir den Bibliotheksmodul:

Type.specs[19] ≡

```
$/Type/Typing.gnrc:inst
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich im Modlib-Dokument über die Grundlagen der Typanalyse und die Symbolrollen des `Typing`-Moduls.

4.1 Typeigenschaften

Typen sind Programmobjekte und werden durch `Keys` repräsentiert: `intType` und `realType`. Der ausgezeichnete Wert `NoKey` repräsentiert den unbekannten Typ in fehlerhaften Konstrukten.

Die vordefinierten Bezeichner `intKey` und `realKey` sind Notationen für diese Typen. Der `Typing`-Modul verlangt, daß dies durch die Eigenschaften `IsType` und `Defer` spezifiziert wird. (Damit ist es möglich, die Sprache auch um Typdefinitionen zu erweitern, was hier noch nicht ausgenutzt wird.)

DeclType.pdl[20] ≡

```
intType -> IsType = {1};
realType -> IsType = {1};

intKey -> Defer = {intType};
realKey -> Defer = {realType};
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich im PDL-Dokument über die Bedeutung von "known keys" und deren Spezifikation.

E-Aufgabe: Erweitern Sie die Sprache um den vordefinierten Typ `bool`.

Die `Kind`-Eigenschaft repräsentieren wir auch durch `Keys`, weil sie bei Spracherweiterungen allgemeiner verwendbar sind als z. B. eine Codierung durch ganze Zahlen. Eine `Kind`-Eigenschaft für Typen brauchen wir nicht vorzusehen, da sie durch die Eigenschaft `IsType` von anderen Programmobjekten unterschieden werden:

DeclKind.pdl[21] ≡

```
Kind: DefTableKey;
VarKind;
ConstKind;
```

This macro is attached to a product file.

Wir führen Attributnamen für die beiden Eigenschaften ein und ordnen die Wurzelrolle des Moduls der Grammatikwurzel zu. (Wir überschreiben die Berechnung von `GotDefer`, da sie stören würde, weil unsere Sprache keine Typdefinitionen hat.)

TypingUse.lido[22] ≡

```
ATTR Type: DefTableKey;
ATTR Kind: DefTableKey;

SYMBOL Program INHERITS RootType COMPUTE
   SYNT.GotDefer = "yes";
END;
```

This macro is attached to a product file.

Der Modul `Typing` stellt Symbolrollen zur Verfügung, die das Setzen und Lesen von Typ- und anderen Eigenschaften vereinfachen:

Eine Deklaration ordnet allen darin definierten Objekten die gleichen `Type`- und `Kind`-Eigenschaften zu. Diese Aufgabe erledigen die Rollen `TypedDefinition` und `TypedDefId`. Wir brauchen nur die Attribute von `Declaration` passend zu bestimmen:

DeclType.lido[23] ≡

```
SYMBOL Declaration INHERITS TypedDefinition END;
```

```
RULE: Declaration ::= DefIdents ':' TypeDenoter ';' COMPUTE
  Declaration.Type = TransDefer(TypeDenoter.Type)
    <- INCLUDING Program.GotDefer;
  Declaration.Kind = VarKind;
END;

RULE: Declaration::= DefIdent '=' IntLiteral ';' COMPUTE
  Declaration.Type = intType;
  Declaration.Kind = ConstKind;
END;

SYMBOL DefIdent INHERITS TypedDefId COMPUTE
  INH.GotProp = ResetKind (THIS.Key, INCLUDING Declaration.Kind);
END;
```

This macro is attached to a product file.

Die Symbolrollen ordnen die `Type`-Eigenschaft selbständig zu. Alle weiteren Eigenschaften (hier: `Kind`) mußsen explizit gesetzt werden. Die Nachbedingung `GotProp` zeigt an, daß dies geschehen ist.

I-Aufgabe: Wie könnte man die Berechnungen formulieren, wenn noch eine weitere Eigenschaft gesetzt werden müßte? Hinweis: LIDO-Funktion `ORDER`.

Der Modul sorgt dafür, daß an Anwendungsstellen diese Eigenschaften und die `Type`-Eigenschaft verfügbar ist. Dies erledigen die Modulrollen `TypeDefUseId` für Typbezeichner und `TypedUseId` für andere Bezeichner:

UseType.lido[24] ≡

```
SYMBOL UseIdent INHERITS TypedUseId, ChkTypedUseId COMPUTE
  SYNT.Kind = GetKind (THIS.Key, VarKind) <- THIS.Type;
END;

SYMBOL UseTypeIdent INHERITS TypeDefUseId, ChkTypeDefUseId END;
RULE: TypeDenoter ::= UseTypeIdent COMPUTE
  TypeDenoter.Type = UseTypeIdent.Type;
END;
```

This macro is attached to a product file.

Die Modulrollen `ChkTypedUseId` und `ChkTypeDefUseId` übernehmen die Prüfung, daß Typbezeichner als Typangaben in Deklarationen und nur dort auftreten.

I-Aufgabe: Schlagen Sie im `Typing`-Dokument die Beschreibung der hier verwendeten Modulrollen nach.

4.2 Typprüfung

Wir müssen die Einhaltung folgender Regeln prüfen:

1. Auf der linken Seite von Zuweisungen und in `input`-Anweisungen müssen Variable stehen.

2. Der Typ des Ausdrucks einer Zuweisung muß mit dem der Variablen verträglich sein.

3. Die Typen von Operanden müssen mit den entsprechenden Typen der Signatur des Operators verträglich sein. (Dies wird im Abschnitt "Operatoren in Ausdrücken" weiter unten behandelt.)

4. Gleiche Typen sind verträglich und `int` ist mit `real` verträglich.

Wir spezifizieren die Typbestimmung und -prüfung in Ausdrücken und Anweisungen nach folgender Strategie: `Type`-Attribute werden in den unteren Kontexten von `Expression` berechnet. Wir berechnen ein Attribut `RequiredType` als den im oberen `Expression` erwarteten Typ, um die Kompatibilität im `Expression`-Symbolkontext zu prüfen. Die Funktion `TypeCompatibleTo` wird weiter unten spezifiziert.

CompatType.lido[25] ≡

```
ATTR RequiredType: DefTableKey;

SYMBOL Expression COMPUTE
  IF (NOT (TypeCompatibleTo (THIS.Type, THIS.RequiredType)),
  message (ERROR, "wrong type of expression", 0, COORDREF));
END;
```

This macro is attached to a product file.

Für einfache Ausdrücke werden die Typen als `Type`-Attribute wie folgt bestimmt:

SimpleType.lido[26] ≡

```
RULE: Expression ::= UseIdent COMPUTE
  Expression.Type = UseIdent.Type;
END;

RULE: Expression ::= IntLiteral COMPUTE
  Expression.Type = intType;
END;

RULE: Expression ::= FloatLiteral COMPUTE
  Expression.Type = realType;
END;
```

This macro is attached to a product file.

In Zuweisungen und `input`-Anweisungen wird die `Kind`-Eigenschaft geprüft. Der `Expression.RequiredType` wird passend bestimmt:

ExprTypeCheck.lido[27] ≡

```
RULE: Statement ::= UseIdent ':=' Expression ';' COMPUTE
  Expression.RequiredType = UseIdent.Type;

  IF (NE (UseIdent.Kind, VarKind),
  message (ERROR, CatStrInd ("assignment must be to a variable: ",
                             UseIdent.Sym),
           0, COORDREF));
END;
```

```
RULE: Statement ::= 'input' UseIdent ';' COMPUTE
  IF (NE (UseIdent.Kind, VarKind),
  message (ERROR, CatStrInd ("input must read into a variable: ",
                             UseIdent.Sym),
           0, COORDREF));
END;

RULE: Statement ::= 'output' Expression ';' COMPUTE
  Expression.RequiredType = Expression.Type;
END;
```

This macro is attached to a product file.

I-Aufgabe: Erläutern Sie die Bedeutung der Berechnung in der `output`-Anweisung.

E-Aufgabe: Erweitern Sie die Sprache um `if`- und `while`-Anweisungen. Müssen Sie Typprüfungen ergänzen?

Die Funktion `TypeCompatibleTo` wird als C-Modul definiert.

`NoKey` repräsentiert den unbekannten Typ in fehlerhaften Konstrukten. Er wird als kompatibel mit jedem Typ angesehen.

I-Aufgabe: Wie würde es sich auswirken, wenn die Vergleiche mit `NoKey` weggelassen würden?

TypeCheck.c[28] ≡

```
#include "TypeCheck.h"
#include "pdl_gen.h"

#ifdef PROTO_OK
int TypeCompatibleTo (DefTableKey narrow, DefTableKey wide)
#else
int TypeCompatibleTo (narrow, wide) DefTableKey narrow, wide;
#endif
{
  return
    (narrow == NoKey || wide == NoKey ||
     narrow == wide ||
       (narrow == intType && wide == realType)
    );
}
```

This macro is attached to a product file.

Die Schnittstelle des C-Moduls wird durch folgende Datei beschrieben:

TypeCheck.h[29] ≡

```
#ifndef TYPECHECK_H
#define TYPECHECK_H
#include "deftbl.h"
#include "eliproto.h"
```

```
#ifdef PROTO_OK
extern int TypeCompatibleTo (DefTableKey narrow, DefTableKey wide);
#else
extern int TypeCompatibleTo ();
#endif
#endif
```

This macro is attached to a product file.

Die Schnittstelle des C-Moduls wird verfügbar gemacht durch:

TypeCheck.head[30] ≡

```
#include "TypeCheck.h"
```

This macro is attached to a product file.

4.3 Operatoren in Ausdrücken

Eine spezielle Aufgabe der Typbestimmung ist die Identifikation überladener Operatoren. Operatoren wie `*` können entweder die Multiplikation von `int`-Werten oder von `real`-Werten bedeuten. Die Unterscheidung wird anhand der Typen der Operanden getroffen. `-` ist der überladene Quelloperator, die `int`-Subtraktion ist ein Zieloperator. Die Zieloperatoren haben unterschiedliche Signaturen: die `int`-Subtraktion verknüpft zwei `int`-Werte und liefert einen `int`-Wert.

Der Ergebnistyp des identifizierten Zieloperators bestimmt den Typ eines Ausdrucks mit dem Quelloperator. Außerdem bestimmt der Zieloperator die Übersetzung des Ausdrucks.

Zur Lösung dieser Aufgabe verwenden wir den Modul `Operator`.

Operator.specs[31] ≡

```
$/Type/Operator.gnrc +referto=(Operator.d) :inst
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich in der Modlib-Dokumentation über den Operator-Modul.

In der Datei `Operator.d` werden die Quell- und Zieloperatoren durch Makroaufrufe definiert.

Operator.d[32] ≡

Quelloperatoren[33]
Zieloperatoren[34]

This macro is attached to a product file.

Zu jedem Quelloperator werden das Operatorzeichen, das Symbol der konkreten und das der abstrakten Syntax, sowie ein eindeutiger Name des Quelloperators angegeben. Der Operator-Modul ergänzt unter anderem auch die konkrete Produktion für den Operator.

Quelloperatoren[33] ≡

```
SrcOpr ('+',  AddOpr, BinOpr, AddSrcOpr)
SrcOpr ('-',  AddOpr, BinOpr, SubSrcOpr)
SrcOpr ('*',   MulOpr, BinOpr, MulSrcOpr)
```

```
SrcOpr ('/',   MulOpr, BinOpr, DivSrcOpr)
```

This macro is invoked in definition 32.

I-Aufgabe: Machen Sie sich mit der Bedeutung der einzelnen Argumente des Macros `SrcOpr` vertraut. Ermitteln Sie in der Implementierung des Moduls, welche Spezifikationsteile durch den Aufruf des Operators hinzugefügt werden.

Für die Zieloperatoren werden der Name ihres Quelloperators, ein eindeutiger Name des Zieloperators, seine Signatur und seine Übersetzung angegeben. Da Operatoren überladen sind, haben verschiedene Zieloperatoren mit unterschiedlicher Signatur den gleichen Quelloperatornamen:

Zieloperatoren[34] ≡

```
TgtOpr(AddSrcOpr, iAddOpr, (intType,intType):intType, TgtStr={"+"})
TgtOpr(AddSrcOpr, rAddOpr, (realType,realType):realType, TgtStr={"+"})
TgtOpr(SubSrcOpr, iSubOpr, (intType,intType):intType, TgtStr={"-"})
TgtOpr(SubSrcOpr, rSubOpr, (realType,realType):realType, TgtStr={"-"})
TgtOpr(MulSrcOpr, iMulOpr, (intType,intType):intType, TgtStr={"*"})
TgtOpr(MulSrcOpr, rMulOpr, (realType,realType):realType, TgtStr={"*"})
TgtOpr(DivSrcOpr, iDivOpr, (intType,intType):intType, TgtStr={"/"})
TgtOpr(DivSrcOpr, rDivOpr, (realType,realType):realType, TgtStr={"/"})
```

This macro is invoked in definition 32.

I-Aufgabe: Machen Sie sich mit der Bedeutung der einzelnen Argumente des Macros `TgtOpr` vertraut. Ermitteln Sie in der Implementierung des Moduls, welche Spezifikationsteile durch den Aufruf des Operators hinzugefügt werden.

Die Übersetzung der Operatoren verwenden wir erst später. Sie wird als PDL-Eigenschaft der Zieloperatoren spezifiziert, die wir schon hier angeben, damit wir den Transformationsteil der Spezifikation abtrennen können: **TransOperator.pdl**[35] ≡

```
TgtStr:  CharPtr; "Strings.h"
```

This macro is attached to a product file.

In der Spezifikationssprache für Oil geben wir an, daß `int`-Werte implizit in `real`-Werte konvertiert werden können:

Coercion.oil[36] ≡

```
COERCION cFloat (intType): realType;
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich in der Oil-Dokumentation über die Bedeutung des `COERCION`-Konstrukts.

In den Kontexten von Ausdrücken mit Operatoren werden die Operandentypen am Operatorsymbol zusammengefaßt. Die Berechnungen zur Identifikation der Zieloperatoren werden von Rollen der Operator-Moduls bezogen. Die Signatur des Zieloperators steht dann als Attribute `LTType`, `RTType` und `ResType` am Operatorsymbol zur Verfügung.

OperatorType.lido[37] ≡

```
SYMBOL BinOpr INHERITS BinTgtOpr, ChkOpr END;
```

```
RULE: Expression ::= Expression BinOpr Expression COMPUTE
  BinOpr.LType = Expression[2].Type;
  BinOpr.RType = Expression[3].Type;
  Expression[1].Type = BinOpr.ResType;
  Expression[2].RequiredType = BinOpr.LTType;
  Expression[3].RequiredType = BinOpr.RTType;
END;
```

This macro is attached to a product file.

I-Aufgabe: Begründen sie, weshalb alle Berechnungen zur Operatoridentifikation am `BinOpr`-Kontext zusammengefaßt sind und nicht im `Expression`-Kontext formuliert werden.

I-Aufgabe: Welche Prüfung erledigt die `ChkOpr`-Rolle? Welche Ergebnisse erwarten Sie daraus für diese (noch nicht erweiterte) Sprache?

E-Aufgabe: Erweitern Sie Core um Vergleichsoperatoren, logische Operatoren und unäres Minus.

5 Transformation in C-Programme

Die Transformation wird nach folgender Methode gelöst:

Für die verschiedenen Formen der Programmkonstrukte `Declaration`, `Statement` und `Expression` wird ihre Übersetzung als PTG-Muster entworfen. Die dazu generierten Funktionen werden in `RULE`- oder `SYMBOL`-Kontexten zur Berechnung von Attributen namens `Ptg` mit dem Typ `PTGNode` verwendet. Die `Ptg`-Attribute werden entweder im darüberliegenden Kontext direkt zur Erzeugung größerer Strukturen eingesetzt oder im Falle von Listen durch `CONSTITUENTS`- oder `CHAIN`-Konstrukte zusammengefaßt.

Diese Dokument spezifiziert nur die Erzeugung eines Programmrahmens. Sein Rumpf ist noch leer. Die Aufgaben leiten Sie an, die Übersetzung zu vervollständigen.

5.1 Programmrahmen

Im `Program`-Kontext wird das Zielprogramm (`Source.Ptg`) auf eine Datei ausgegeben, deren Name aus dem Namen der Quelldatei durch Anhängen von `.c` gebildet wird:

TransProgram.lido[38] ≡

```
ATTR Ptg: PTGNode;

RULE: Program ::= Source COMPUTE
  PTGOutFile (CatStrStr (SRCFILE, ".c"), Source.Ptg);
END;

RULE: Source ::= 'program' Block COMPUTE
  Source.Ptg = Block.Ptg;
END;
```

This macro is attached to a product file.

Die Funktion `CatStrStr` haeb wir schon im Analyseteil importiert. Die Variable für den Namen der Quelldatei importieren wir aus folgendem C-Modul:

TransProgram.head[39] ≡

```
#include "source.h"
```

This macro is attached to a product file.

Der äußere (zunächst einzige) Block bildet den Rahmen des C-Programms, in den die Übersetzungen der Deklarationen und der Anweisungen eingebettet werden. Er wird durch das PTG-Muster `FrameBlock` spezifiziert:

Frame.ptg[40] ≡

```
FrameBlock:
        "#include <stdio.h>\n"
        $1 /* declarations */
        "void main () {\n"
        $2 /* statements */
        "exit (0);\n}\n"
```

This macro is attached to a product file.

Es besteht aus einer Folge von Zeichenreihen, in die an den mit `$1` und `$2` bezeichneten Stellen die Texte für Deklarationen bzw. Anweisungen eingesetzt werden. Sie werden beim Aufruf der Funktion `PTGFrameBlock` als Parameter übergeben.

I-Aufgabe: Informieren Sie sich im PTG-Dokument über die Schreibweise von PTG-Mustern und die Bedeutung der `$i`-Markierungen darin.

Für die folgenden Spezifikationen benutzen wir einen Modul, der oft verwendete PTG-Muster enthält:

TransConstDecl.specs[41] ≡

```
$/Output/PtgCommon.fw
```

This macro is attached to a product file.

I-Aufgabe: Informieren Sie sich im Modlib-Dokument über die Muster, die der Modul verfügbar macht.

Die Deklarationen und Anweisungen des Blockes werden in den Programmrahmen eingefügt.

E-Aufgabe: Wenn Sie Core um geschachtelte Blöcke erweitern, müssen innere Blöcke anders übersetzt werden als der äusserste Block. Wie lösen sie dies?

BlockTrans.lido[42] ≡

```
RULE: Block ::= 'let' Declarations 'in' Statements 'end'
COMPUTE
  Block.Ptg =
    PTGFrameBlock (Declarations.Ptg, Statements.Ptg);
END;

CHAIN ConstDeclPtg, VarDeclPtg: PTGNode;
SYMBOL Declarations COMPUTE
  CHAINSTART HEAD.ConstDeclPtg = PTGNULL;
  CHAINSTART HEAD.VarDeclPtg = TAIL.ConstDeclPtg;
```

```
  SYNT.Ptg = TAIL.VarDeclPtg;
END;

CHAIN StmtPtg: PTGNode;
SYMBOL Statements COMPUTE
  CHAINSTART HEAD.StmtPtg = PTGNULL;
  SYNT.Ptg = TAIL.StmtPtg;
END;
```

This macro is attached to a product file.

Alle Konstantendeklarationen werden den Variablendeklarationen vorangestellt. (Dies ist eigentlich nicht erforderlich, es dient hier nur dazu, um die Technik zu demonstrieren.)

Aufgabe: Wie wird dieser Effekt durch obige Berechnungen erreicht? Informieren Sie sich im LIDO-Dokument über die Verwendung von `CHAIN`s.

Mit den übersetzten Deklarationen und Anweisungen wird die Funktion `PTGFrameBlock` aufgerufen. (Diese Berechnung ist als Symbolberechnung formuliert, damit sie bei Spracherweiterungen überschrieben werden kann.)

5.2 Deklarationen

Konstantendeklarationen werden in initialisierte Variablendeklarationen übersetzt:

TransConstDecl.ptg[43] ≡

```
ConstDecl:      "int " $1 /* ident */ " = " $2 /* value */ ";\n"
```

This macro is attached to a product file.

Hier, wie in den folgenden Abschnitten werden nur an den Knoten Berechnungen spezifiziert, die Texte zu einer der `CHAIN`s beitragen. Sie werden mit dem bis zu dem Knoten berechneten `CHAIN`Wert durch die Funktion `PTGSeq` konkateniert.

TransConstDekl.lido[44] ≡

```
RULE: Declaration ::= DefIdent '=' IntLiteral ';' COMPUTE
  Declaration.ConstDeclPtg = PTGSeq (Declaration.ConstDeclPtg,
        PTGConstDecl (DefIdent.Ptg, PTGNumb (IntLiteral)));
END;
```

This macro is attached to a product file.

In Variablendeklarationen werden der Grundtyp und die Deklaratoren für jeden Bezeichner eingesetzt.

TransVarDecl.ptg[45] ≡

```
VarDecl:        $1 /* base type */ " " $2 /* declarators */ ";\n"
```

This macro is attached to a product file.

Die Bezeichner einer Variablendeklaration sammeln wir mit einer `CHAIN` auf, die für jede `Definition` lokal ist.

TransDeclarators.lido[46] ≡

```
CHAIN DeclaratorsPtg: PTGNode;

SYMBOL Declaration COMPUTE
  CHAINSTART HEAD.DeclaratorsPtg = PTGNULL;
END;

SYMBOL DefIdent COMPUTE
  THIS.DeclaratorsPtg =
    PTGCommaSeq (THIS.DeclaratorsPtg, THIS.Ptg);
END;
```

This macro is attached to a product file.

Aufgabe: Erläutern Sie die Verwendung des `CommaSeq`-Musters.

Typangabe und Bezeichnerliste werden in das `VarDecl`-Muster eingefügt. Um für `real` Typen den C-Bezeichner `float` zu erzeugen, verwenden wir die Funktion `PTGAsIs`. (Sie wird duch den unten benutzten Bibliotheksmodul verfügbar gemacht.)

TransVarDecl.lido[47] ≡

```
RULE: Declaration ::= DefIdents ':' TypeDenoter ';' COMPUTE
  Declaration.VarDeclPtg = PTGSeq (Declaration.VarDeclPtg,
        PTGVarDecl (TypeDenoter.Ptg, DefIdents.DeclaratorsPtg));
END;

SYMBOL TypeDenoter COMPUTE
  SYNT.Ptg =
    IF (EQ (INCLUDING Declaration.Type, realType),
      PTGAsIs ("float"),
      CONSTITUENT UseTypeIdent.Ptg);
END;
```

This macro is attached to a product file.

5.3 Anweisungen

Zuweisungen werden mit folgenden Mustern übersetzt:

TransAssignStmt.ptg[48] ≡

```
AssignStmt:     $1 /* lhs */ " = " $2 /* rhs */ ";\n"
```

This macro is attached to a product file.

TransAssignStmt.lido[49] ≡

```
RULE: Statement ::= UseIdent ':=' Expression ';' COMPUTE
  Statement.StmtPtg = PTGSeq (Statement.StmtPtg,
        PTGAssignStmt (UseIdent.Ptg, Expression.Ptg));
END;
```

This macro is attached to a product file.

input- und output-Anweisungen werden mit folgenden Mustern übersetzt:

TransInOutStmt.ptg[50] ≡

```
InputIntStmt:   "scanf (\"%d\", &(" $1 /* variable */ "));\n"
InputRealStmt:  "scanf (\"%e\", &(" $1 /* variable */ "));\n"

OutPutIntStmt:  "printf (\"%d\\n\","  $1 /* expression */ ");\n"
OutPutRealStmt: "printf (\"%e\\n\","  $1 /* expression */ ");\n"
```

This macro is attached to a product file.

Anhand der Variablentypen entscheiden wir zwischen den beiden Varianten.

TransInOutStmt.lido[51] ≡

```
RULE: Statement ::= 'input' UseIdent ';' COMPUTE
  Statement.StmtPtg = PTGSeq (Statement.StmtPtg,
        IF (EQ (UseIdent.Type, intType),
                PTGInputIntStmt (UseIdent.Ptg),
                PTGInputRealStmt (UseIdent.Ptg)));
END;

RULE: Statement ::= 'output' Expression ';' COMPUTE
  Statement.StmtPtg = PTGSeq (Statement.StmtPtg,
        IF (EQ (Expression.Type, intType),
                PTGOutPutIntStmt (Expression.Ptg),
                PTGOutPutRealStmt (Expression.Ptg)));
END;
```

This macro is attached to a product file.

5.4 Bezeichner und Literale

Texte für Bezeichner und Literale werden durch Bibliotheksmodul PtgCommon erzeugt. Dieses Modul wurde bereits in Abschnitt 5.2 referenziert. Dies braucht hier nicht wiederholt zu werden.

TransToken.lido[52] ≡

```
SYMBOL SamePtg COMPUTE SYNT.Ptg = PTGId (TERM); END;

SYMBOL DefIdent         INHERITS SamePtg END;
SYMBOL UseIdent         INHERITS SamePtg END;
SYMBOL UseTypeIdent     INHERITS SamePtg END;
```

This macro is attached to a product file.

Aufgabe: Informieren Sie sich in der Modlib-Dokumentation über den PtgCommonModul.

Die folgenden Berechnungen reichen die Ptg-Attributwerte identisch nach oben im Baum weiter:

TransExpr.lido[53] ≡

```
RULE: Expression ::= UseIdent COMPUTE
```

```
    Expression.Ptg = UseIdent.Ptg;
  END;

  RULE: Expression ::= IntLiteral COMPUTE
    Expression.Ptg = PTGNumb (IntLiteral);
  END;

  RULE: Expression ::= FloatLiteral COMPUTE
    Expression.Ptg = PTGId (FloatLiteral);
  END;
```

This macro is attached to a product file.

Aufgabe: Warum werden hier Gleitpunktzahlen mit `PTGId` übersetzt?

5.5 Operatoren

Ausdrücke werden in der Übersetzung vollständig geklammert, da die Präzedenzregeln in Core und C unterschiedlich sind.

TransExpr.ptg[54] ≡

```
  BinExpr:        "(" $1 /*lopd */ $2 string /* opr */ $3 /* ropd */ ")"
```

This macro is attached to a product file.

Der Operator-Modul berechnet den Zieloperator als `TgtKey`-Attribut des Operatorsymbols. Die Eigenschaft `TgtStr` des Zieloperators gibt die Übersetzung an, die in der Spezifikation `Operator.d` festgelegt ist.

TransOpr.lido[55] ≡

```
  RULE: Expression ::= Expression BinOpr Expression COMPUTE
    Expression[1].Ptg =
          PTGBinExpr (Expression[2].Ptg,
                      GetTgtStr (BinOpr.TgtKey, "??"),
                      Expression[3].Ptg);
  END;
```

This macro is attached to a product file.

I-Aufgabe: Überlegen Sie, wie man bestimmte Zieloperatoren (z. B. Potenzierung) in Aufrufe von C-Funktionen übersetzen könnte.