Records in IML

Compilerbau HS 2015, Team BB
Team: Livio Bieri, Raphael Brunner

Schlussbericht vom 09.01.2015

Abstract

Die Erweiterung **Records in IML** wurden als eigener Typ `record` umgesetzt. *Der Compiler ist in Swift geschrieben. Den Sourcecode findet man auf Github.*

Beschreibung der Erweiterung

Die Erweiterung soll sogenannte **Records** (auch bekannt als *struct* oder *compound data*)[1] zur Verfügung stellen. Ein Record soll dabei als neuer Datentyp zur Verfügung stehen. Er soll beliebig viele Felder beinhalten können *(mindestens jedoch eins)*. Felder können vom Datentyp Integer, Boolean oder Record sein.

Eine **Deklaration** in IML sieht wie folgt aus:

- `var example: record(x: int32, b: boolean)`

Der **Zugriff** ist wie folgt möglich:

- `debugin example.x`
- `debugout example.x`
- Die Felder können vom Datentyp `boolean`oder `int32`sein `record`. *Nested Records sind vorgesehen.*

[1] https://en.wikipedia.org/wiki/Record_(computer_science)

Beispiel

```
program main
global
    var position: record ( var x:int32, const y:int32 ) ;
    professor: record ( id:int32, level:int32 )

do
    position.x init := 4;
    position.y init := 5;

    debugout position.x;
    debugin position.x;

    debugout position.x;

    professor.id init := 1007;
    professor.level init := 19;

    position.x := 42;
    position.x := 5 * (position.y + 5) + 1;
    debugout position.x;

    debugout professor.id

endprogram
```

Funktionalität und Typeinschränkung

Deklaration des Record

Die Deklaration eines Records kann im `global` aber auch lokal vorgenommen werden:

```
...
global
    var position: record(x: int32, y: int32);
    const professor: record(id: int32, level: int32)
do
...
```

Eindeutigkeit des Record Identifier

Die Deklaration (der *Identifier*) eines Records muss eindeutig sein:

```
var position: record(x: int32, y: int32);
const position: record(z: int32, u: int32) // Fehler
      ^^^^^^^^
```

Zu beachten ist, dass dies aber natürlich generell gilt:

```
var position: int32;
var position: record(x: int32, y: int32); // Fehler
    ^^^^^^^^
```

Eindeutigkeit der Record Felder Identifier

Die Deklaration eines Record Felds muss eindeutig sein:

```
var position: record(x: int32, x: int32) // Fehler
                               ^
```

Felder Eindeutigkeit muss aber nur innerhalb *eines Records gegeben sein:*

```
var positionXY: record(x: int32, y: int32);
var positionXYZ: record(x: int32, y: int32, z: int32)
```

Typenchecking (`bool`, `int32`, `record`)

Der zugewiesene Wert muss vom Typ sein, der in der Deklaration angegeben wurde (`bool`, `int32` oder `record`):

```
var point: record(x: int32, y: int32);

point.x := true; // Fehler
           ^^^^
```

Zugriff auf undefinierte Felder

Der Zugriff auf Felder, die nicht definiert wurden, ist nicht möglich:

```
var point: record(x: int32, y: int32);

point.z = 42; // Fehler
      ^
```

Unterstützung von `CHANGEMODE` in Records

Records unterstützen `CHANGEMODE` (`var`, `const`):

- `CHANGEMODE` ist optional.
- Falls nicht angegeben, wird `const` verwendet.

```
point: record(x: int32, y: int32)
```

Wird interpretiert als:

```
const point: record(const x: int32, const y: int32)
```

Felder unterstützen ebenfalls `CHANGEMODE`, wenn jedoch der ganze Record `const` ist, dürfen seine Felder nicht `var` sein:

```
const point: record(const x: int32, var y: int32) // Fehler
                                    ^^^
```

Operationen auf Records:

Records selbst haben *keine Operationen. Folgendes ist also nicht möglich / wird nicht unterstützt:*

```
pointZero: record(var x: int32, var y: int32);
pointOne: record(var x: int32, var y: int32)
...

pointZero = pointZero + pointOne // Fehler
                      ^
```

Vergleich mit anderen Sprachen

Wir haben uns unterschiedliche Lösungsansätze angeschaut. Dazu haben wir uns vor allem angeschaut, was andere Sprachen konkret machen:

Haskell

Deklaration:

```
data vector = vector {
    x::Int, y::Int, z::Int}
```

Value Constructor (Function)

```
v1 = vector 5 6 7
```

Pascal

Deklaration eines Types `TVector`:

```
type TVector = record
    x : Integer ;   y : Integer ;   z : Integer ;
end ;
```

Initialisierung einer Variable vom Type `TVector`:

```
var v1 : TVector
begin
    v1.x := 42;
    v1.y := 50;
    v1.z := 20;
end ;
```

C

Deklaration eines `struct vector`:

```
struct vector {
    int x;
    int y;
    int z;
};
```

Initialisierung einer Variable vom Type `vector`:

```
vector v1;
v1.x = 42;
v1.y = 50;
v1.z = 20;
```

IML

Deklaration:

```
var v1: record(x: int32, y: int32, z: int32)
```

Initialisierung:

```
v1.x init := 42;
v1.y init := 42;
v1.z init := 42
```

Einfluss auf unsere Lösung:

- Unser Ziel war es eine IML-ähnliche Syntax beizubehalten.
- Unsere Spezifikation orientiert sich lose an der Pascal Spezifikation.

Lexikalische und grammatikalische Syntax

- Unser Ziel ist es, das Record ähnlich wie die anderen Variablen in IML zu behandeln.
- Daher wird die Initialisierung eines Records analog zur Initialisierung der Variablen stattfinden.

Die Grundgrammatik-Idee eines Records für die Initialisierung:

```
Im Folgenden gilt:  Esp = Epsilon
```

```
    optRecordDeclaration ::= LPAREN recordDecl RPAREN | Eps
    recordDecl           ::= storageDeclaration repRecordFields
    repRecordFields      ::=
      COMMA storageDeclaration repRecordFields | Eps
    optionalCHANGEMODE   ::= CHANGEMODE | Eps
    storageDeclaration   ::= optionalCHANGEMOD typeIdent
    typeIdent            ::= IDENT COLON typeDeclaration
    typeDeclaration      ::= TYPE optRecordDeclarationList
    optionalCHANGEMODE   ::= CHANGEMODE | Eps
```

`storageDeclaration` wird im globalen Raum deklariert und somit werden Records gleich wie die normalen Variablen behandelt. Sie haben einen eigenen TYPE der mit einem RECORD Token repräsentiert wird.

Zugriffe auf die Werte in einem Record sollen in die Expression Grammatik eingefügt werden, damit wir uns nicht separat mit den Problemen wie `Debugin` oder `Debugout` beschäftigen müssen.

```
expression                  ::= term1 BOOLOPRterm1
BOOLOPRterm1                ::= BOOLOPR term1 BOOLOPRterm1 | Eps
term1                       ::= term 2 RELOPRterm2
RELOPRterm2                 ::= RELOPR term2 RELOPRterm2 | Eps
term2                       ::= term3 ADDOPRterm3
ADDOPRterm3                 ::= ADDOPR term3 ADDOPRterm3 | Eps
term3                       ::= term4 MULTOPRterm4
MULTOPRterm4                ::= MULTOPR term4 MULTOPRterm4 | Eps
term4                       ::= factor DOTOPRfactor
DOTOPRfactor                ::= DOTOPR IDENT | Eps
factor                      ::= LITERAL
                                | IDENT optionalIInitFuncSpec
                                | LPAREN expression RPAREN
optionalIInitFuncSpec       ::= INIT | expressionList | Eps
expressionList              ::= LPAREN optionalExpressions RPAREN
optionalExpressions         ::= expression
```

```
                                    repeatingOptionalExpressions | Eps
repeatingOptionalExpressions ::= COMMA expression
                                    repeatingOptionalExpressions | Eps
```

Compiler

Wir haben unseren IML Compiler in Swift programmiert. Insgesamt waren wir sehr zufrieden mit unserer Entscheidung. Vor allem das Konzept der Optionals / bzw. des Optional Chaining aber auch Pattern Matching stellte sich als äusserst praktisch heraus. Einzig die Anbindung an die Virtuelle Maschine gestaltete sich schwierig (da in Java).

Scanner

Grundsätzlich war die nötige Erweiterung für Records im Scanner sehr einfach. Es musste lediglich ein neues 'Keyword' definiert werden in `keywords.swift`. Es handelt sich dabei um einen `Type`.

```
...
"record": Token(
    terminal:
        Terminal.TYPE,
    attribute:
        Token.Attribute.Type(Token.TypeIdentifier.RECORD)
),
...
```

CST

Die Erweiterung im CST war hauptsächlich, dass die `Typedeclaration` um ein optionales Feld `optionalRecordDecl` erweitert wurde, welches falls es sich um Record handelt (anhand des `type` erkennbar) dieses Feld gesetzt hat.

```
    class TypeDeclaration: ASTConvertible {
        let type : Token.Attribute
        let optionalRecordDecl : OptionalRecordDeclaration?
        ...
    }
```

Der Record selbst enthält dann die RecordFields:

```
    class RecordDecl: ASTConvertible {
        let storageDeclartion: StorageDeclaraction
        let repeatingRecordFields: RepeatingRecordFields?
        ...
    }
```

AST

Beim AST verhält es sich ähnlich wie beim CST, da wurde lediglich das RecordDecl weg abstrahiert:

```
    class DeclarationStore: Declaration {
        let changeMode: ChangeMode?
        let typedIdent: TypeDeclaration
        let nextDecl: Declaration?
        ...
```

```
    class TypeDeclaration: Declaration {
        let ident: String
        let type: Token.Attribute
        let optionalRecordDecl: DeclarationStore?
        ...
```

Wir bauen dann die Record Felder rekursiv über DeclarationStore auf, da TypeDeclaration auch ein Kind von DeclarationStore ist, kann man so theoretisch beliebig viele nested Records in Records deklarieren.

Checker

Beim Context Check mussten wir für die Records einige Ausnahmen bilden. Mit dem eingeführtem Dot-Operator müssen wir ebenfalls speziell verfahren im Gegensatz zu den Standard-Operatoren:

```
class DyadicExpr: AST.Expression {
func check(side:Side) throws -> (ValueType, ExpressionType) {
...

if(typeL == ValueType.RECORD) {
    if(oldScope != nil) {
        AST.scope =
      oldScope?.recordTable
        [(expression as! StoreExpr).identifier]!.scope
    } else {
```

```
        AST.scope =
      AST.globalRecordTable
        [(expression as! StoreExpr).identifier]!.scope
    }
}

if let r = term as? StoreExpr {
    if(typeL == ValueType.RECORD && side == Side.LEFT) {
        checkR = try! r.check(.LEFT)
    } else {
        checkR = try! r.check(.RIGHT)
    }
} else if let r = term as? DyadicExpr {
    checkR = try! r.check(.RIGHT)
} else {
    checkR = try! term.check()
}
...

case .DotOperator:
    valueSide = .L_Value
    if(typeL == ValueType.RECORD){
        let lhs = expression as! StoreExpr
        let rhs = term as! StoreExpr

        let leftIdent: String = lhs.identifier
        let rightIdent: String = rhs.identifier

        let identifier: String = leftIdent + "." + rightIdent

        if(AST.scope != nil){
            guard let type =
        AST.scope!.storeTable[identifier]?.type else {
                    throw ContextError.SomethingWentWrong
            }
            expressionType = type
        } else {
            guard let type =
        AST.globalStoreTable[identifier]?.type else {
                    throw ContextError.SomethingWentWrong
            }
            expressionType = type
        }
    } else {
        throw ContextError.TypeErrorInOperator
```

```
    }
case _:
    throw ContextError.SomethingWentWrong
}
...
```

Hier setzen wir dann die linke und rechte Seite vom operator zu einem neuen Identifier zusammen, der jeweils die Seiten mit einem "." trennt. Da Punkte in der normalen Namen nicht erlaubt sind, müssen wir so keine Kollisionen mit anderen Identifier rechnen. Auch müssen wir ein init auf der Rechten Seite eines Operators zulassen, da die Syntax folgendes unterstützen muss: `recordName.recordField init := 4`
Bei der `StoreExpr` müssen wir darauf achten, dass der Recordidentifier nicht direkt initialisiert werden kann, sondern dass dies nur mit seinen Feldern geht:

```
class StoreExpr: Expression {
...
func check(side:Side) throws -> (ValueType, ExpressionType) {
...
if(initToken != nil) {
    if(side == Side.RIGHT){
        throw ContextError.InitialisationInTheRightSide
    }
    if(expressionType == ValueType.RECORD) {
        throw ContextError.RecordCanNotBeInitializedDirectly
    }
    if(store.initialized) {
        throw ContextError.IdentifierAlreadyInitialized
    }
    store.initialized = true
} else if(side == Side.LEFT && !store.initialized
  && expressionType != ValueType.RECORD){
    throw ContextError.IdentifierNotInitialized
} else if(side == Side.LEFT && store.isConst) {
    throw ContextError.NotWriteable
} else if(side == Side.RIGHT && !store.initialized) {
    throw ContextError.IdentifierNotInitialized
}
...
```

Weiter unterscheiden wir in `DeclarationStore` ob es sich dabei um einen Record handelt. Falls es ein Record ist, erstellen wir einen Record im Context und prüfen die Type Felder und speichern sie entsprechend im aktuellen Scope ab, dabei wird der Identifier mit dem Recordfeld- und dem Recordnamen gebildet:

```
if(type == ValueType.RECORD){
    let record = Record(ident: typedIdent.ident)
    let recordStore = Store(
    ident: record.ident,
    type: type,
    isConst: isConst)
    recordStore.initialized = true

    var decl:DeclarationStore? = typedIdent.optionalRecordDecl!

    if(AST.scope != nil){
        if let _ = AST.scope!.recordTable[record.ident] {
            throw ContextError.IdentifierAlreadyDeclared
        } else {
            AST.scope!.recordTable[record.ident] = record
            AST.scope!.storeTable[record.ident] = recordStore
        }
    } else {
        if let _ = AST.globalRecordTable[record.ident] {
            throw ContextError.IdentifierAlreadyDeclared
        } else {
            AST.globalRecordTable[record.ident] = record
            AST.globalStoreTable[record.ident] = recordStore
        }
    }

    while(decl != nil){
        let store:Store = try! decl!.check()

        let oldIdent = store.ident
        store.ident = recordStore.ident + "." + store.ident
        record.scope.storeTable[oldIdent] = store

        if(isConst && store.isConst != isConst){
            throw ContextError.RecordIsConstButNotTheirFields
        }
        if(AST.scope != nil){
            if let _ = AST.scope!.storeTable[store.ident] {
                throw ContextError.IdentifierAlreadyDeclared
            } else {
                AST.scope!.storeTable[store.ident] = store
                record.recordFields[store.ident] = store
            }
        } else {
            if let _ = AST.globalStoreTable[store.ident] {
```

```
                throw ContextError.IdentifierAlreadyDeclared
            } else {
                AST.globalStoreTable[store.ident] = store
                record.recordFields[store.ident] = store
            }
            store.adress = AST.allocBlock++
            print("alloc: \(store.ident), adress: \(store.adress)")
        }

        decl = decl!.nextDecl as? AST.DeclarationStore
    }
...
```

Codegeneration

Alle AST Klassen haben eine Funktion `code(let loc:Int) throws -> Int` welcher den Code für die entsprechende Klasse generiert. Dieser ruft dann rekursiv weiter die Funktion bei seinen Kindern auf. Dabei wird `loc` mitgegeben, welcher die Codelocation repräsentiert. Damit können wird dann das `codeArray` mit der richtigen Position befüllen.

Ähnlich wie beim AST, mussten wir für Records jeweils immer eine Sonderausnahme implementieren. Diese sind jedoch von der Art her überall ähnlich. Wir müssen Prüfen ob der Identifier der gerade gefragt ist ein Record ist, falls ja ersetzen wir unseren Record durch seine Felder und lassen dann den Code generieren:

```
class ExpressionList: AST {
...
func code(let loc:Int) throws -> Int {
    var loc1:Int = loc
    if let expr = expression as? StoreExpr {
        let store:Store
        if(AST.scope != nil){
            store = AST.scope!.storeTable[expr.identifier]!
        } else {
            store = AST.globalStoreTable[expr.identifier]!
        }
        if(store.type == ValueType.RECORD){
            let record:Record
            if(AST.scope != nil){
                record = AST.scope!.recordTable[expr.identifier]!
            } else {
                record = AST.globalRecordTable[expr.identifier]!
            }
```

```
            for (_, field) in record.recordFields {
                loc1 = field.code(loc1)
            }
        } else {
            loc1 = try! expression.code(loc1)
        }
    } else {
        loc1 = try! expression.code(loc1)
    }
    guard let newLoc = try! optExpression?.code(loc1) else {
        return loc1
    }
    return newLoc
}
...
```

Virtualmachine

Da wir nicht direkt von Swift mit unserer Virtuellen Machine kommunizieren können, haben wir ein kleines CLI Interface programmiert, welches uns erlaubt die virtuelle Maschine (das `CodeArray`) via `System.in` zu steuern. Das sieht dann etwa so aus:

```
# pipes code generated by compiler to vm
cat code.intermediate | java machine
```

Wobei der Code in `code.intermediate` wie folgt vorliegt:

```
...
2, AllocBlock 4,
3, LoadImInt 0,
4, InputInt m,
...
```

Siehe auch:

- `VirtualMachine/src/Machine.java`, *CLI Interface*
- `VirtualMachine/src/vm/CodeArray.java`, `fromSystemIn(...)`

Offene Punkte

Die folgenden Punkte konnten wir bis zur Abgabe leider nicht komplett lösen:

- Mehrere Dyadic-Operationen in einer Expression führen zu Fehlermeldungen.
- Nested Records konnte leider nicht implementiert werden.
- Funktionen funktionieren nicht.

Appendix

- **Sourcecode & Dokumentation: https://github.com/livioso/cpib**
- *Arbeitsteilung: Wir haben die Arbeiten wie folgt im Team verteilt. Bieri: Scanner, Zwischenbericht, CST, AST, VM, Codegeneration, Schlussbericht / Brunner: Grammatik (SML), Zwischenbericht, AST, Checker, Codegeneration, Schlussbericht.*

Ehrlichkeitserklärung

Hiermit erklären wir, dass wir die vorliegenden Bericht und den Compiler selbständig verfasst bzw. programmiert haben. Eine Zusammenarbeit mit anderen Teams fand nicht statt.

Ort / Datum / Unterschrift

Livio Bieri

Raphael Brunner