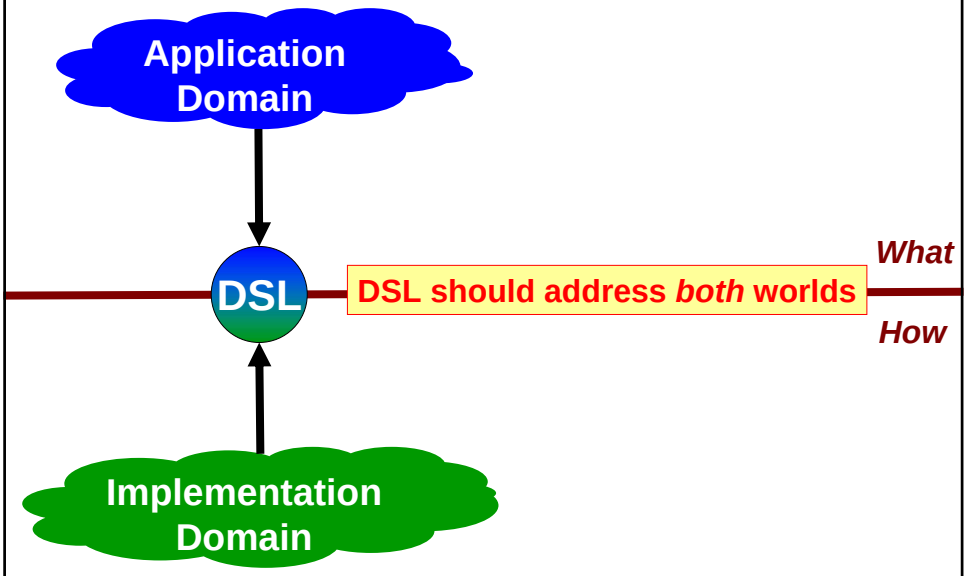
WG 2.11
Dagstuhl Meeting 2006

Type Systems to Support Language Integration

Jörg Striegnitz

Domain Specific Knowledge Comes From two Ends



Language and Thoughts

Wilhelm von Humboldt (1767 – 1845):

Language is the *building organ* of thoughts

(Sprache ist das bildende Organ der Gedanken)

Edward Sapir (1884 – 1934):

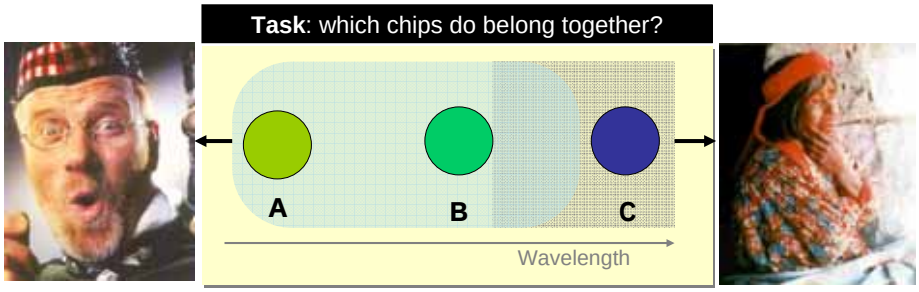
Language is *structuring* thoughts

Benjamin Lee Whorf (1897 – 1941):

Language *determines* thoughts

Evidence:
- **Hopi … o words to express time**
- **The … tive in Chinese**
- **Esk… words for snow**

The Kay-Kampton Experiment



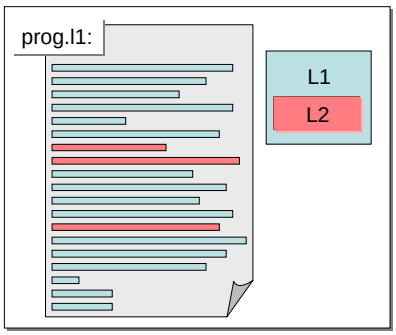
English: „*blue*“ and „*green*“

Taramuhara: „*grue*“

Conclusion: language ***may influence problem perception*** and the quality of the ***solution***!

Approaches to Integrate Different Languages

- **Special purpose languages**
  - Multiparadigm languages (e.g. Curry, Babel, Leda)
- **Multilingual Systems**
  - Foreign Function Interfaces (e.g. JNI)
  - Virtual Machines (e.g. JVM, .NET)
- **Language Embeddings**
  - Libraries
  - Interpreters (and staged interpreters)



Examples for Embeddings

SQL embedded in JAVA:

Interpreter approach

```
Resultset rs = stat.executeQuery("select * from Adressen");
while (rs.next) {
  String s    = rs.getString("Name");
  Integer plz = rs.getInteger(“ZIP");
  System.out.println(s + " " + n);
}
```

Library used in C:

Library approach

```
Col_vector_double x,z;
Line_vector_double  y;
Matrix_double  md;

md = mult_MD_CVD(mult_CVD_LVD(x, y),z);
…
```

	DSL *Syntax*	DSL *Integration*	DSL *Typechecking*	DSL *Optimization*
Interpreter approach			Within DSL Marshalling	
Library approach				

Examples for Embeddings

SQL embedded in JAVA:

Interpreter approach

Resultset rs = stat.executeQuery("select * from Adressen");
while (rs.next) {
String s = rs.getString("Name");
Integer plz = rs.getInteger(“ZIP");
System.out.println(s + " " + n);
}

Library used in C:

Library approach

Matrix_double md;

md = mult_MD_CVD(mult_CVD_LVD(x, y),z);
…

	DSL *Syntax*	DSL *Integration*	DSL *Typechecking*	DSL *Optimization*
Interpreter approach			Within DSL Marshalling	
Library approach				

Language Integration © 2006 Jörg Striegnitz 8

How to combine these approaches?

Examples for Embeddings

SQL embedded in JAVA:

Interpreter approach

```
Resultset rs = stat.executeQuery("select * from Adressen");
while (rs.next) {
  String s    = rs.getString("Name");
  Integer plz = rs.getInteger("ZIP");
  System.out.println(s + " " + n);
}
```

C++ Expression Templates

Library approach

```
Col_vector<double> x,z;
Line_vector<double>  y;
Matrix<double>  md;

md = x*y*z;
…
```

	DSL *Syntax*	DSL *Integration*	DSL *Typechecking*	DSL *Optimization*
Interpreter approach			Within DSL / Marshalling	
Library approach				

Library Approach: How to Optimize the Syntax

DSL-Program hard to read:

```
md = mult_MD_SVD(mult_SVD_ZVD(x, y),z);
```

Desirable: Overloading:

```
md = mult(mult(x, y), z);
```
or
```
md = x*y*z;
```

To support extensibility, **`mult`** must have a very common type:

$$\texttt{mult} :: \forall\alpha.\forall\beta.\forall\gamma.\ \alpha \times \beta \rightarrow \gamma \qquad (\alpha,\beta,\gamma \text{ arbitrary types})$$

Problems

Typing:
- Programmer must be able to constrain a function's type (e.g. **`mult(Matrix<n,m>, Matrix<n,m>)`** for **`n≠m`**)
- Compiler must know result type

⇒ **Programmer manipulates / extends *type system* [Typecase]**

Code generation:
- Heterogeneous translation of overloaded functions!
- Optimizations?

⇒ **Programmer manipulates / extends *translation process* [typecase]**

Library Approach: How to Optimize the Syntax

DSL-Program hard to read:

```
md = mult_MD_SVD(mult_SVD_ZVD(x, y),z);
```

Desirable: Overloading:

```
md = mult(mult(x, y), z);   or   md = x*y*z;
```

To support extensibility, **`mult`** must have a very common type:

`mult :: ∀α.∀β.∀γ. α × β → γ` (α,β,γ arbitrary types)

Problems

Typing:
- Programmer must be able to constrain a function's type (e.g. `mult(Matrix<n,m>, Matrix<n,m>)` for n≠m)
- Con
- ⇒ Prog [typecase]

Code and result type depend on argument types

Code generation:
- Heterogeneous translation of overloaded functions!
- Optimizations?
- ⇒ Programmer manipulates / extends *translation process* [typecase]

Types Depending on Types

Programmer can extend typing rules:

```
*T[T]:= Typecase T of
          default:                          TypeError();
          Matrix<n,m> × Matrix<m,n>:        Matrix<n,n>
          SqMatrix<T1> × SqMatrix<T2>:      SqMatrix<*T[T1× T2]>
          α × β:                            IF α≡β then β
                                            else IF α <:β then β
                                                 else …
```

Code Depending on Types

Programmer can generate type-dependent code:

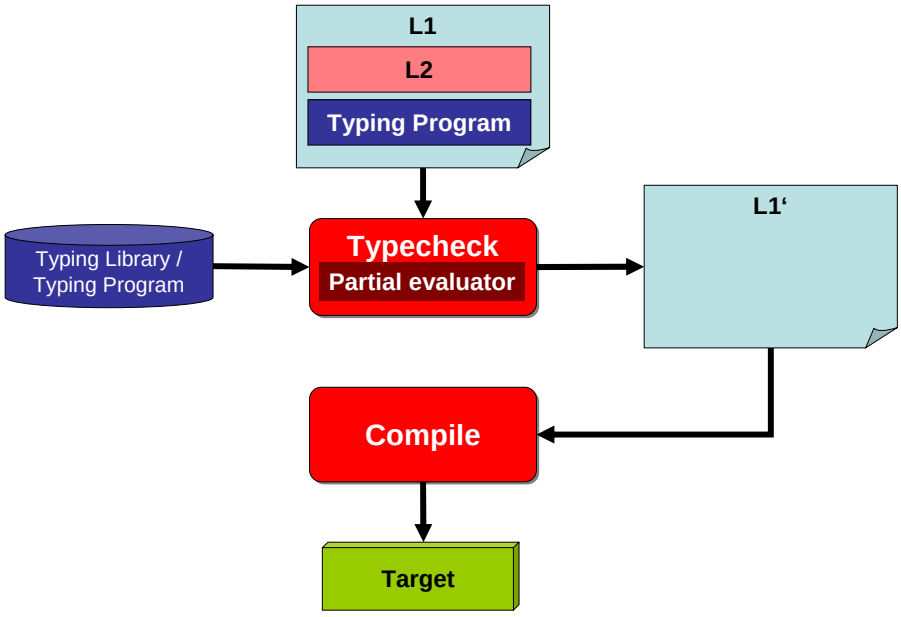
```
*V[T]:= typecase T of
  Matrix<n,m> × Matrix<m,n>: f(x,y):=matmult(x,y)
  SqMatrix<T1> × SqMatrix<T2>:
    f(M1,M2):= {
                SqMatrix< *T[T1× T2] > r;
                for (i=0;i<x.size;++i)
                  for (j=0;j<y.size;++j)
                    for (k=0;k<x.size;++k)
                      r[i,j] += *V[T1× T2](M1[j,k]*M2[k,i]);
                return r;
               }
              …
```

Multiplication: $*_{\mathbb{V}}\,[T_1 \times T_2]\,(t_1, t_2)$

Overload-application: $*_{\mathbb{V}}\llbracket t_1, t_2 \rrbracket$

Even more syntax sugar: $t_1 *_{\mathbb{V}} t_2$

Translation Process



Polymorphic Constructors

… to construct types and values

- Built-in tpyes (e.g. `int`, `float`, `string`, `void`) are nullary constructors (aka *constants*)
- nullary constructors are ***values*** and ***types*** at the same time:

$$\vdash \texttt{int} : \texttt{int} \quad \vdash \texttt{float} : \texttt{float} \quad \vdash \texttt{S} : \texttt{S} \quad \cdots$$

- *n*-ary type constructors / value constructors:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \cdots \quad \Gamma \vdash t_n : T_n}{\Gamma \vdash C(t_1, \cdots, t_n) : C(T_1, \cdots, T_n)}$$

- **These rules describe an initial Σ-term algebra!**

Use this algebra to represent embedded program

A very simple example …

V	::=	V-Terme
	x	Variable
	N	Ganze Zahl
	add(V,V)	Addition
	mul(V,V)	Multiplikation

```
num(x)   = Num(x)
add(x,y) = Add(x,y)
mul(x,y) = Mul(x,y)
```

Objects of the Σ-term algebra

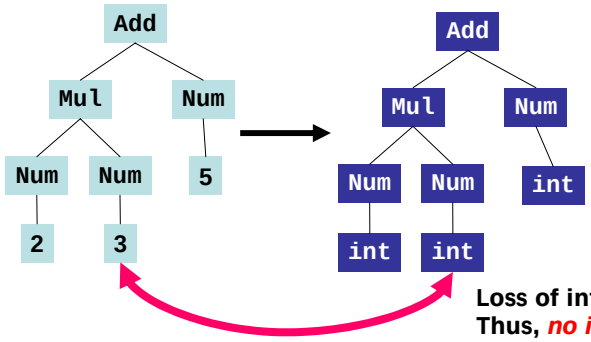
Functions of the host language
(elements of the embedded language)

```
      add(mul(num(3),num(5)),num(7))
⟶    Add(Mul(Num(3),Num(5)),Num(7))
```

Type Derivation for Add(Mul(2,3),5)

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash 2 : \texttt{int}}{\Gamma \vdash \texttt{Num}(2) : \texttt{Num}(\texttt{int})} \quad \dfrac{\Gamma \vdash 3 : \texttt{int}}{\Gamma \vdash \texttt{Num}(3) : \texttt{Num}(\texttt{int})}}{\Gamma \vdash \texttt{Mul}(\texttt{Num}(2),\texttt{Num}(3)) : \texttt{Mul}(\texttt{Num}(\texttt{int}),\texttt{Num}(\texttt{int}))} \quad \dfrac{\Gamma \vdash 5 : \texttt{int}}{\Gamma \vdash \texttt{Num}(5) : \texttt{Num}(\texttt{int})}}{\Gamma \vdash \texttt{Add}(\texttt{Mul}(\texttt{Num}(2),\texttt{Num}(3)),\texttt{Num}(5)) : \texttt{Add}(\texttt{Mul}(\texttt{Num}(\texttt{int}),\texttt{Num}(\texttt{int})),\texttt{Num}(\texttt{int}))}$$

Notice: *structural equivalence* of type and value!



Loss of information at the leafs
Thus, *no isomorphism*

A V-Interpreter

```
vInter[T](t)  :=  typecase T of
                    Num:  t.0
                    Mul:  vInter[T.0](t.0) * vInter[T.1](t.1)
                    Add:  vInter[T.0](t.0) + vInter[T.1](t.1)
                                  ...

vInter[  add(mul(num(2),num(3)),num(5))  ]
```

Problems

- **num** not part of embedded language (V)
- Type safety of V is lost
  (**add**, **mul** and **num** are too general!)
- Explicit call to interpreter needed
- Interpreted code is slow!
- Type-dependent computations at runtime?

```
num(x)   = Num(x)
add(x,y) = Add(x,y)
mul(x,y) = Mul(x,y)
```

Optimization 1: Syntax und Type System

Introduce new function check:

```
check[T](t)  :=  typecase T of
                   default:  TypeError()
                   int:      Num(t)
                   Num:      t
                   Mul:      Mul( check[T.0](t.0), check[T.1](t.1))
                   Add:      Add( check[T.0](t.0), check[T.1](t.1))
```

Adaptation of **add** and **mul**:

$$\texttt{add}[T_1,T_2](t_1,t_2) = \texttt{check}\,[\texttt{Add}(t_1,t_2)]$$
$$\texttt{mul}[T_1,T_2](t_1,t_2) = \texttt{check}\,[\texttt{Mul}(t_1,t_2)]$$

Consequences

- Embedded language has to use overload-application

$$\texttt{add}\,[\texttt{mul}\,[2,3]\,,5]$$

- Type errors get detected now:

$$\texttt{add}\,[\texttt{mul}\,[\text{"}\mathit{Hello}\text{"},3.5]\,,'c']$$

Optimization 2: Futamuras Projections (1972)

$$[\![P_L]\!]_L \;\; i_1 \;\cdots\; i_n = r$$

If the first *k*<*n* inputs are known, for a correct working partial evaluator **mix** we have:

$$[\![\texttt{mix} \;\; P_L \;\; i_1 \;\cdots\; i_k]\!]_L \;\; i_{k+1} \;\cdots\; i_n = r$$

First Futamura-Projection

Starting point: Interpreter for a language L_E, written in der in L_H

$$[\![\texttt{inter}]\!]_{L_H} \;\; P_{L_E} \;\; i$$

P_{L_E} is *static information* – we can apply **mix**:

$$[\![\texttt{mix inter} \;\; P_{L_E}]\!]_{L_H} \;\; i$$

Translator for L_H exists!

Optimization 2: Futamuras Projections (1972)

Types are static information!

```
check[T](t)  :=  typecase T of
                   default:  TypeError()
                   int:      Num(t)
                   Num:      t
                   Mul:      Mul( check[T.0](t.0), check[T.1](t.1))
                   Add:      Add( check[T.0](t.0), check[T.1](t.1))
```

$$[\![\texttt{inter}]\!]_{L_H}\ P_{L_E}\ i$$

Tree traversal could be done by `mix`!

$$[\![\texttt{mix inter}\ P_{L_E}]\!]_{L_H}\ i$$

Partial Evaluation of check

```
check[T](t)  :=  typecase T of
                   default:  TypeError()
                   int:      Num(t)
                   Num:      t
                   Mul:      Mul( check[T.0](t.0), check[T.1](t.1) )
                   Add:      Add( check[T.0](t.0), check[T.1](t.1) )
```

static type information
static runtime information

$$\texttt{add}\,[\texttt{mul}\,[2,3],5]$$

„de-sugaring“

$$\texttt{add}[\texttt{Add}(\texttt{Mul}(\texttt{int},\texttt{int}),\texttt{int}](\texttt{mul}[\texttt{int},\texttt{int}](2,3),5)$$

`mix`

$$\texttt{Add}(\texttt{Mul}(2,3),5)$$

Partial Evaluating the Interpreter Function

```
vInter[T](t) := typecase T of
                  Num: t.0
                  Mul: vInter[T.0](t.0) * vInter[T.1](t.1)
                  Add: vInter[T.0](t.0) + vInter[T.1](t.1)
```

static type information
static runtime information

$$\texttt{vInter}\,[\texttt{Add}(\texttt{Mul}(2,3),5)]$$

↓ ↓ „de-sugaring“ / `mix`

$$2 * 3 + 5$$

Achievement?

	DSL *Syntax*	DSL *Integration*	DSL *Typechecking*	DSL Optimierung
Interpreter approach				
Library approach				

Conclusion and Future Work

- **Optimizing the embedded language is possible**
  and executed by the partial evaluator
- **Problem: Turing complete language at the type-level**
  Typability no longer decidable!
- **Restricting the type-level language means restricting the expressiveness of the embedded language**
- **Problem: Termination of the partial evaluator**

Three formal languages

Language	Type system	`mix`	DSL
$F^{SA}_{\omega,1}$	decidable	terminates	no recursion simple types
$F^{SA}_{\omega,2}$	decidable	termination not guaranteed	recursion simple types
$F^{SA}_{\omega,3}$	not decidable	termination not guaranteed	recursion complex types

Conclusion and Future Work

- **Optimizing the embedded language is possible**
  and executed by the partial evaluator
- **Problem: Turing complete language at the type-level**
  Typability no longer decidable!
- **Restricting the type-level language means restricting the expressiveness of the embedded language**
- **Problem: Termination of the partial evaluator**

Current work

- **Classify the language that could be embedded**
- **Enhancing the formal languages**
  (e.g. introduce loop-operator, subtyping)
- **Integraion into existing languages**
  (z.B. C#, JAVA, FORTRAN)
- **Excessive overloading**
  (e.g. sequence-operator, assignment, loops)
- **Real world examples**