Hackhofer

DSL in the Insurance business

Motivation
Objectives
TreeCalc Language
Implementation

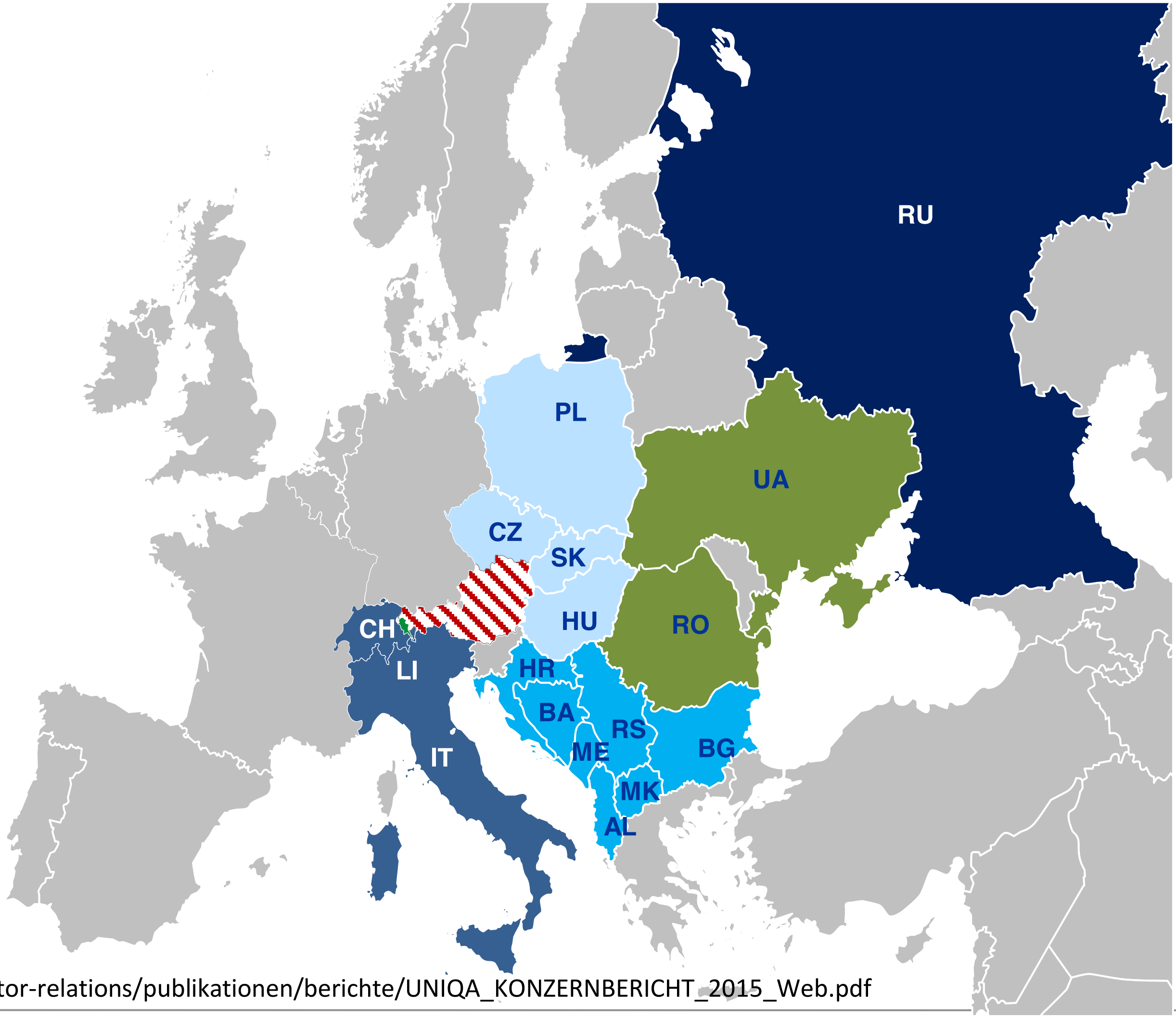
TreeCalc

Hackhofer Software GmbH
1070 Wien
www.hackhofer.com

DI Stefan Neubauer, 29.03.2017

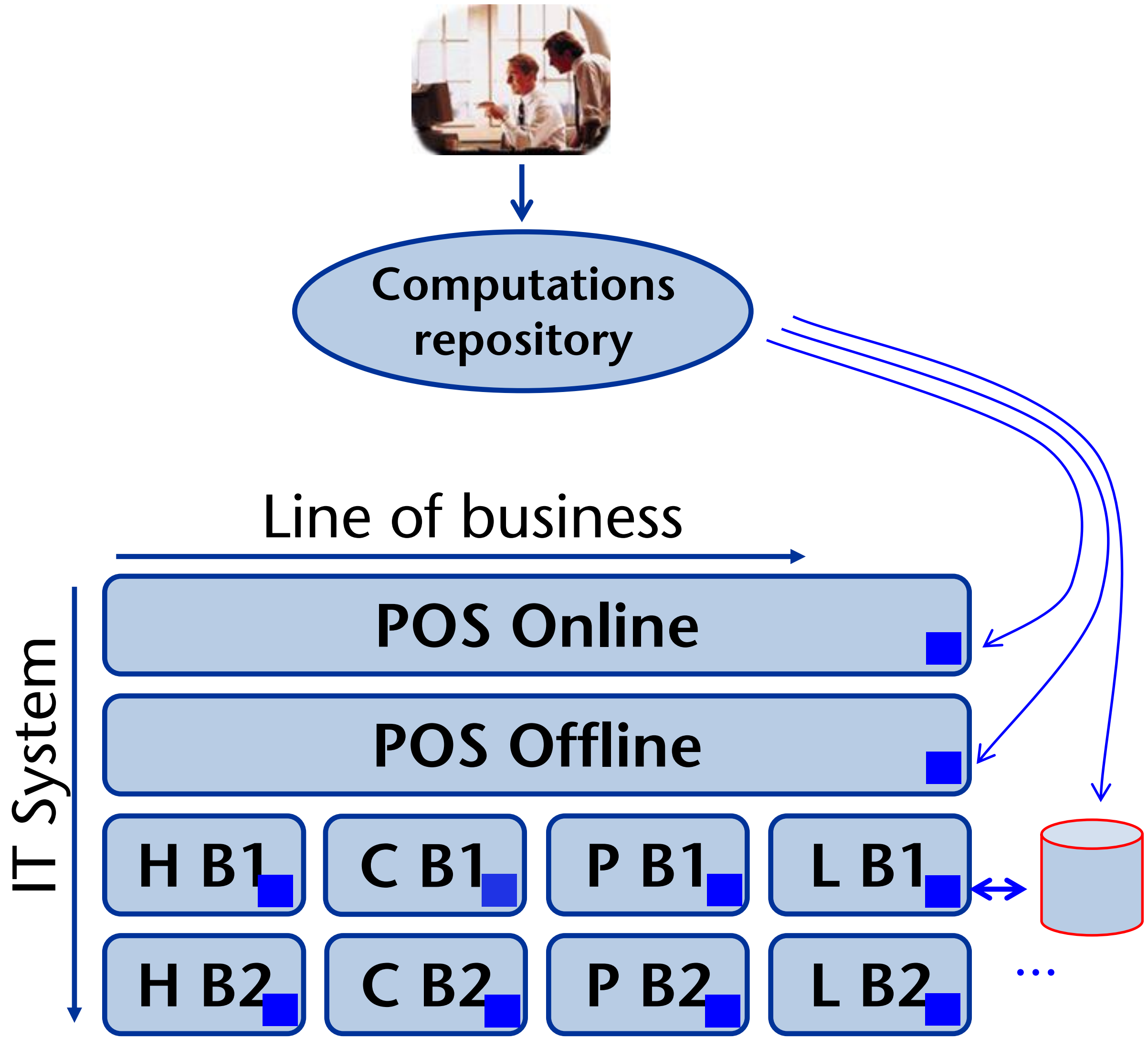
Movitation – Overview UNIQA Group

- 38 insurance companies
- 19 markets
- 14.113 employees
- 10 mn customers
- > 18.6 mn ins. Policies
- Premium volume
  - 6.3 bn EUR
  - 62 % AT, 38 % UI
- Net profit: 199.9 mn EUR



http://www.uniqagroup.com/gruppe/versicherung/investor-relations/publikationen/berichte/UNIQA_KONZERNBERICHT_2015_Web.pdf

Motivation – business view



Hackhofer

Motivation

- VP/MS ® from CSC
  - Main calculation engine for the UNIQA group
  - Proprietary
  - Modelling part: Eclipse based; Actuaries / business units
  - Execution part: Native library (Windows, Linux, z/OS, AS/400)
- TreeCalc
  - Open Source → https://github.com/treecalc
  - Safer, better, faster ☺
  - Playground for Memoization

Objectives

- Language
  - Declarative, „safe“: no destructive assignment, no loops, ..
  - Text format
  - Calculations organized in trees
  - VP/MS models translatable to TreeCalc
- API
  - Simple (set value, compute result, get list, input needed?)
- Execution
  - Optimal integration for: Java, JavaScript
  - Fast, safe, correct, scaleable

DSL implementation options

- Internal DSL / Embedded DSL
  - Host: Lisp, Ruby, (Template) Haskell, (Meta)OCaml, …
  - Fluent interface: Java, C#, ...
  - Highly dependent on host language
- External DSL
  - Custom syntax, custom parsing
  - Semantic model
  - Interpretation / Code generation

Hackhofer

Language - Overview

Result definition
- Tree structure
- Can be combined

Function
- Reusable

Table
- Base values
- List entries

Input definition
- Values set from outside world
- Can be linked to Table
- Definition of UI-properties, ..

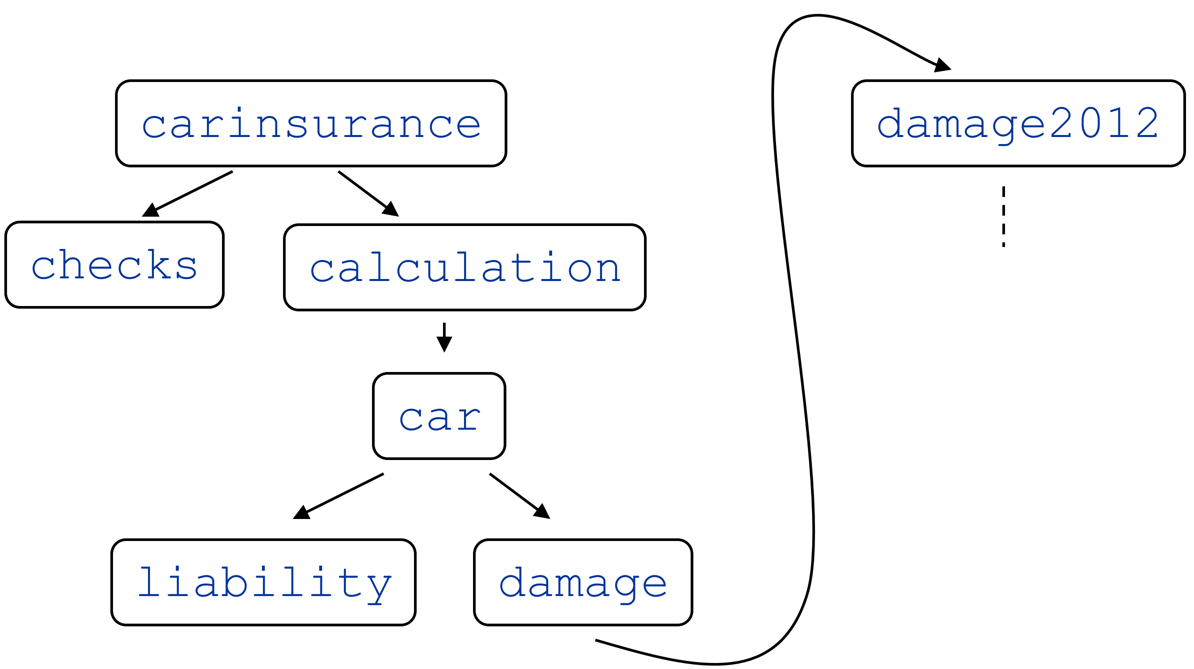
Formulas
- Base functions, table access., ..
- Everything is an expression
- No side effects

Language – Example input defintion

```
INPUT I_DateOfBirth ;

INPUT I_Damage_Sum {
   visible = I_Damage_YN ;
   default = 10000;
   list = T_Damage_Sum ;
   select = key=1 || F_Age(I_DateOfBirth) > 30 ;
}
```

Language – Example tree

```
TREE carinsurance {
     NODE checks ;
     NODE calculation {
          NODE car TIMES I_CarCounter {
               NODE liability ;
               NODE damage IF I_Damage_YN {
                    LINK damage2012;
               }
          }
     }
}
```



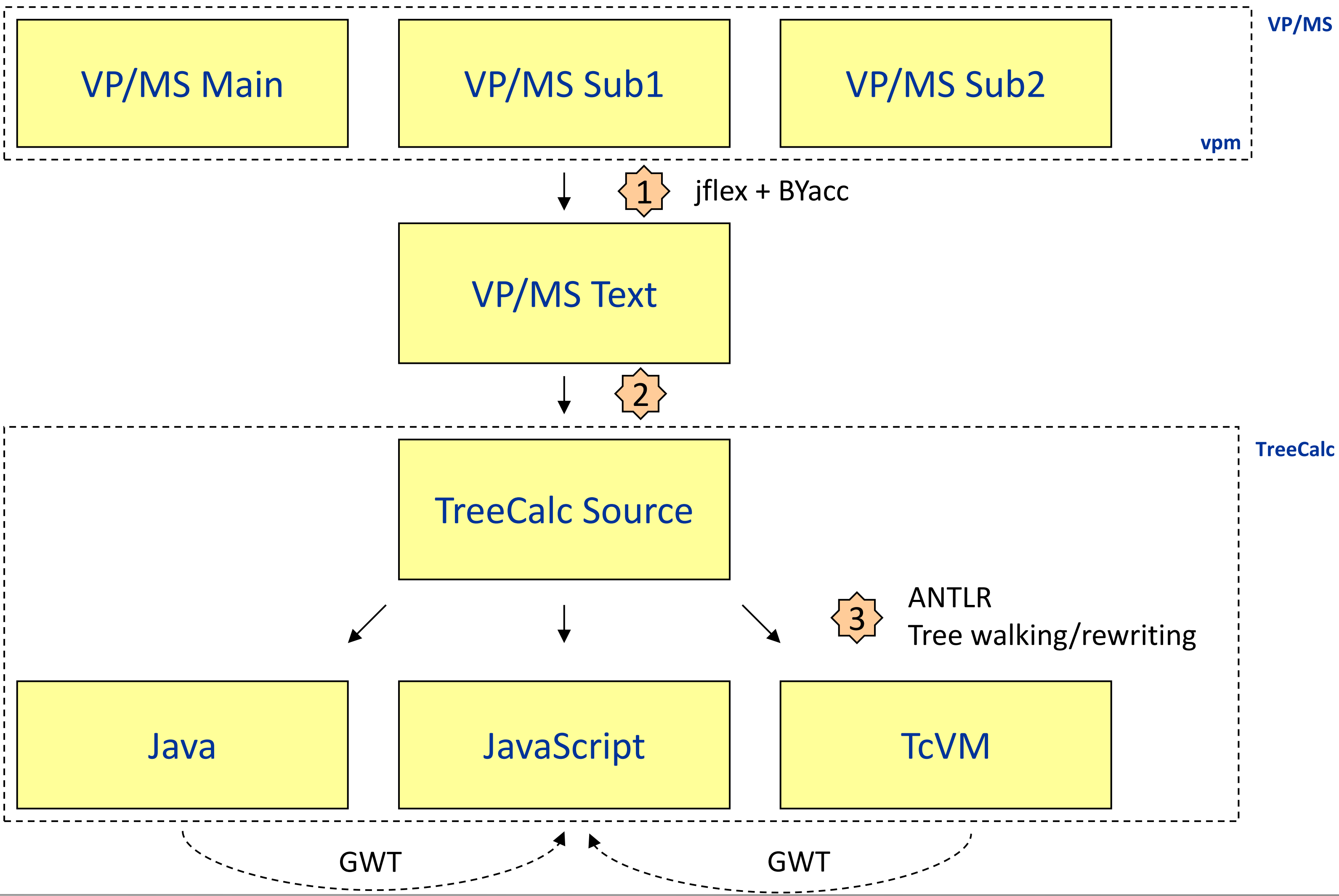
Language – Example result definition

```
CALC carinsurance.calculation {
     RX_Prem = R_Prem
                  *
               IF I_Discount_YN THEN
                    0.8
               ELSE
                    1
               ENDIF ;
}

CALC carinsurance.calculation.car.liability {
     R_Prem = I_kw * T_Area[I_Area].fact ;
     ...
}

CALC damage2012.calculation {
     R_Prem = ...
}
```

Language – Example table definition

```
TABLE T_Mortality (age, qx, qy) {
   16, 0.0006380, 0.0003980 ;
   17, 0.0007200, 0.0004160 ;
   18, 0.0007760, 0.0004060 ;
   19, 0.0008060, 0.0003720 ;
   20, 0.0008400, 0.0003580 ;
    ...
}

TABLE T_Liability_Sum (key, text) {
   1, "€  6.000.000,-" ;
   2, "€ 12.000.000,-" ;
}
```

Implementation – Overview



Implementation – Tools and strategies

- ANTLR
  - Lexer + Parser + Tree construction
  - LL(*), semantic/syntactic predicates
  - Nice grammar, lots of tools, automatic error correction, …
  - Generated code bigger than lex/yacc & co
- Implementation strategies
  - Homogeneous AST
  - External visitor for passes
    1. Scopes
    2. Symbol table
    3. Resolve names + rewrite AST
    4. Fill semantic model

```
void visitNodes(Ast node) {
    switch (node.getType()) {
    case TT_COMPUNIT: {
      List<Ast> c = node.getChildren();
      for (Ast child : c) {
        visitNodes(child);
      }
    break;
    }
    case KEYWORD_TREE: {
     ...
```

Implementation – Code generation

- out.print(...) ☺
- Simple because of semantic model + AST
  - Same model and helper classes
  - Separate writer classes for Java, JavaScript, TcVM
- Formulas
  - intermediate vars _1, _2, ...
  - AST node
    - → optional: out.print(...)
    - → expression string (short expr. or varname)

Implementation - Java

```
F_LI_Lx(age; sex; risk):
  if(age<=0; 100000;
     F_LI_Lx(age-1; sex; risk) * (1 - F_LI_qx(age-1; sex; risk))
  )
```

```
static final V F_LI_LX(S _s, V age, V sex, V risk) {
   Object cacheKey = _s.getCacheKey(8156345, age, sex, risk);
   V ret = _s.readCache(cacheKey);
   if (ret!=null) { return ret; }
   V _1;
   V _2 = age.smleq(_i0);
   if (_2.booleanValue()) {
      _1 = _i100000;
   } else {
      V _3 = age.sub(_i1);
      V _4 = age.sub(_i1);
      V _5 = _i1.sub(F.F_LI_QX(_s, _4, sex, risk));
      V _6 = F.F_LI_LX(_s, _3, sex, risk).mult(_5);
      _1 = _6;
   }
   ret = _1;
   _s.writeCache(cacheKey, ret);
   return ret;
}
```

Implementation - Java highlights

- Performance tweaks
  - final static methods, constant pool, ids instead of names
  - Bit sets for node structure
  - Dynamic dispatch (calc, table, function):
    - switch(id) { ... } instead of reflection
  - Tables: sorted or direct access → O(1) or O(log n)
  - Caching
    - LRU instead of HashMap
    - random id; fast key object creation
    - no caching for simple formulas

- Handicaps
  - 16bit limits → split methods/classes
  - Big code blocks for dynamic dispatch

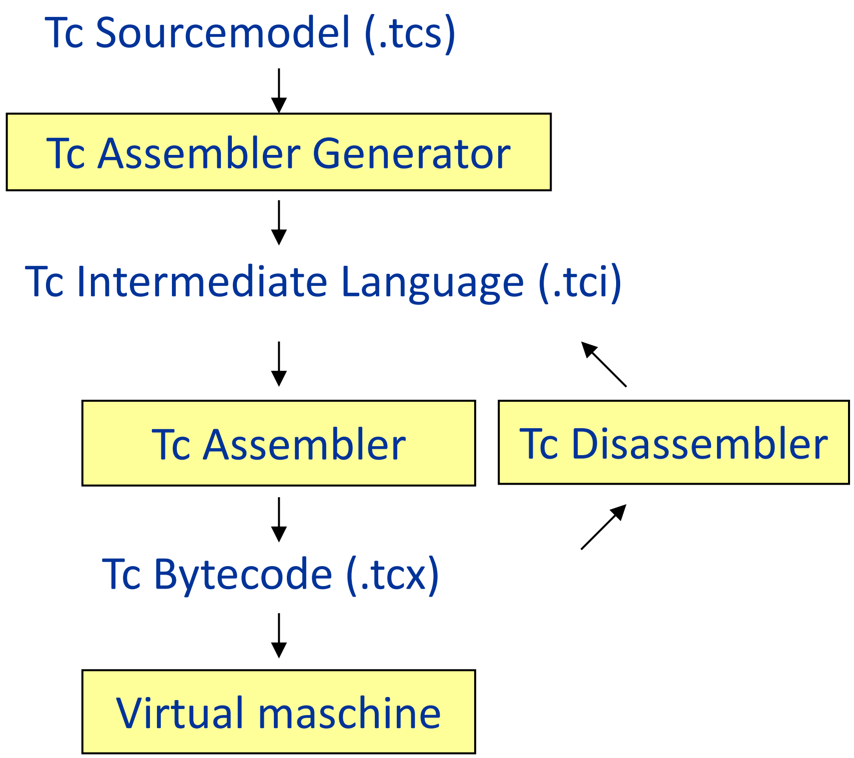
Hackhofer

Implementation - JavaScript

- Some things easier
  - Dynamic dispatch
  - No 16bit limits
- Some things harder
  - Missing base libaries: HashMap, ..
  - Development environment
- Handicaps
  - Large models

Implementation - TcVM

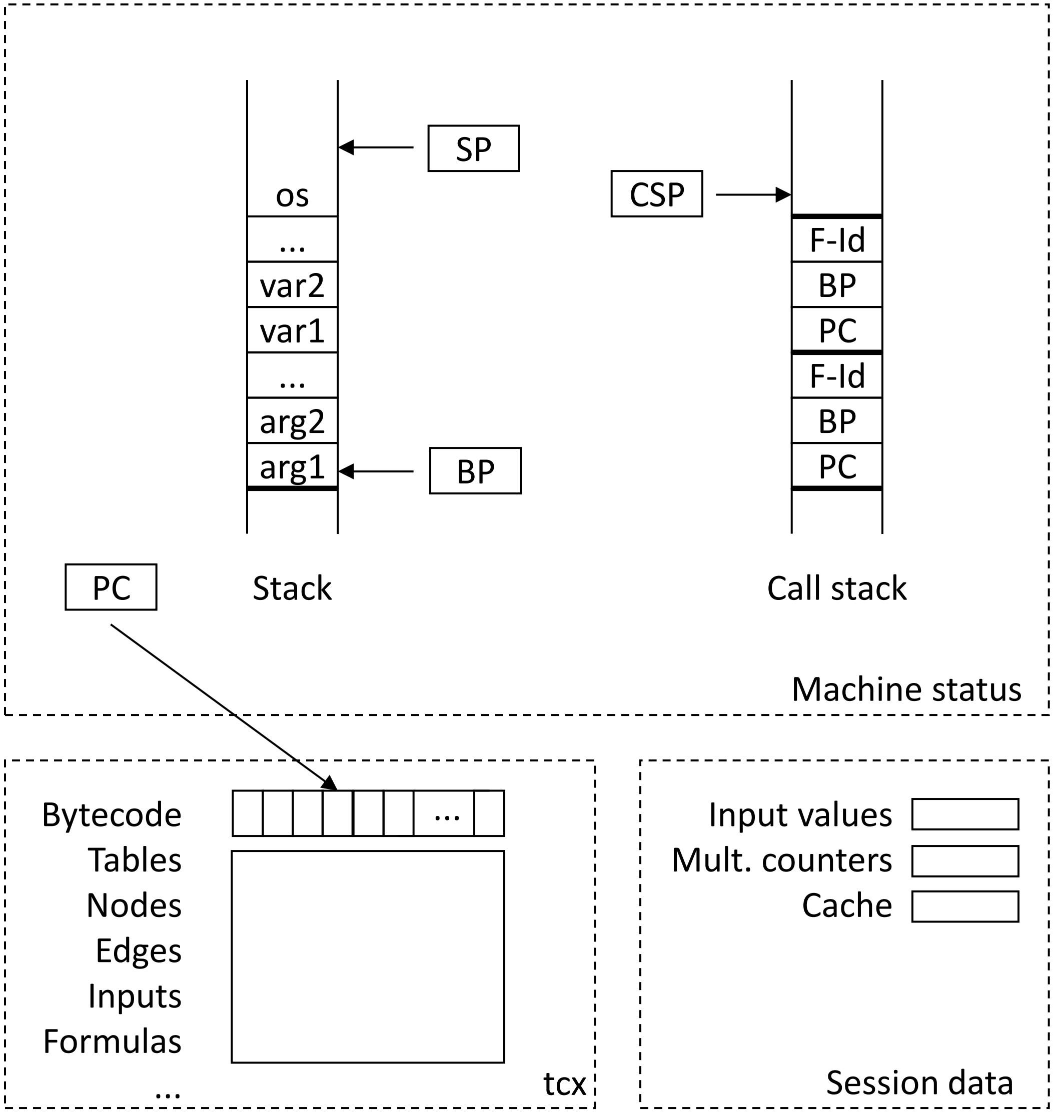
- Why
  - Optimized for size
  - Easy to play around with
- Implementation
  - TcVM implementated in Java
  - For some models too slow



Implementation – TcVM example

```
.func func=1 name=F_FACT args=1 simple=false formula=1
.formula formula=1 ; line 6830
   //start of if statement, line 6830
   : load 0 ; N
   : pushconst 0
   : cmpsmleq
   : iffalse L0
   : pushconst 1
   : goto L1
 L0:
   : load 0 ; N
   : dup
   : pushconst 1
   : sub
   : callfunc 1 ; F_FACT
   : mult
 L1:
   //end of if statement
   : return
```

TcVM - internals



TcVM - highlights

- Numbers
  - Functionality: 100 %
  - Performance: 10 times slower
  - Size: 15 % of generated Java code
- Code generation
  - Text representation (.tci) proved to be useful
  - Generation simpler than Java/JavaScript source code generation
- Implementation
  - Tree access with TcVM „macro programs“ suboptimal
  - Base classes reused (from Java source generated port)

Wish list

- Organization/Community
  - **Build up / extend user group and contributors**
  - Landing page, Wiki, Tutorial, Bug tracker, …
  - Books, Training and Certifications ;-)
- Improvements
  - Test cases + examples
  - Upgrade from ANTLR 3 to ANTLR 4
  - Improve JavaScript port
  - Performance optimizations: Type inference, dynamic memoizaiton, ..
  - Implement features from VP/MS® Runtime XE
- Extensions
  - Partial generation (e.g. just for UI-control)
  - Modelling environment (Xtext, …)
  - VP/MS to TreeCalc converter Open Source

Hackhofer

References

- Domain Specific Languages, Martin Fowler, Addison-Wesley, 2010
- Language Implementation Patterns, Terence Parr, Pragmatic Bookshelf, 2009
- The Definitive ANTLR Reference, Terence Parr, Pragmatic Bookshelf, 2007
- http://www.antlr.org/
- https://github.com/treecalc