Filet-o-Fish
Practical and dependable DSLs for OS development

Pierre-Evariste Dagand* Andrew Baumann Timothy Roscoe

ENS Cachan-Bretagne
ETH Zurich

October 11, 2009
Domain-specific languages for OS development?

Drawback of system languages
- Low-level languages
- Limited power of abstraction
- Weak semantics

Benefit from domain-specific knowledge
- Adapted syntax
- Precise type system
- Powerful optimizations
DSLs for OS development? No, thanks!

Our analysis

- Compiler hard to get right
- Generated code hard to debug
- Fragile semantics

Our proposal

- Importance of semantics
- Mechanized semantics
- Aiming at formally verified DSLs
Overview of a DSL compiler

DSL input → Front-end → AST → Back-end → System code

Front-end
- Parser
- Syntax check
- Type check

Back-end
- AST transformation
- Code generation
- Optimization
Standard back-end construction

Input AST

Transformation

AST'

Code generation

AST''

Pretty-printer

System code

Drawbacks

- Syntactic transformation
- Lose semantics properties
- Hard to recover
Filet-o-Fish approach

- Input AST
- Transformation
- Code generation
- FoF compiler
- System code

FoF as a language
- Defined by a semantics
- Compiled to C

FoF as a DSL library
- Embedded into Haskell
- Integrated in the back-end
Practical DSL construction

Figure: FoF syntax

```
stmt ::= do stmt stmt |
       return expr |
       r ← newRef expr |
       r ← readRef expr |
       writeRef expr expr |
       assert expr |
       ...
```

```
compareCapAddr (parent, addrP, typeP) |
   (child, typeC) field = do
   parentU ← readStruct parent "u" |
   parentCS ← readUnion parentU typeP |
   parentAddr ← readStruct parentCS addrP |
   childU ← readStruct child "u" |
   childCS ← readUnion childU typeC |
   childAddr ← readStruct childCS field |
   returnc (parentAddr .==. childAddr)
```

Figure: Hamlet snippet

Abstracting C inside Haskell

- Clear semantics
- Higher-order programming style
- Compiler generating “valid” C
Beyond syntax

\[ y \leftarrow \text{newRef} \ x \]

**string concatenation**

\[ y \mathbin{++} \ ' = \&, \ ++ \ x \]

**AST manipulation**

\[ \text{ASSIGN} ((\text{VAR} \ y), (\text{ADDR} \ (\text{VAR} \ x))) \]

\[ y \mathbin{++} \ ' = \*, \ ++ \ x \]

\[ \text{ASSIGN} ((\text{ADDR} \ (\text{VAR} \ x)), (\text{VAR} \ y)) \]

**Drawbacks**

- Complete absence of semantics
- Is the generated code “valid”?
- Is the `newRef` semantics preserved?
On-paper semantics

\[ v = E[x] \quad tv = \text{typeof}(v) \]

\[
\begin{align*}
\text{mem,} & \quad \text{heap,} \\
y & \leftarrow \text{newRef } x
\end{align*}
\]

\[
\Rightarrow \begin{align*}
\text{Normal,} & \quad \\
\text{mem}[y \mapsto v], & \quad \text{heap}[y \mapsto \text{pointerT } tv \text{ Available}]
\end{align*}
\]

Drawbacks

- Tedious proofs by hand
- No computer support
- Little reward for this hard work
runNewRef :: Data → Heap → (Loc, Heap)
runNewRef value heap =
  ( CLRef Local typeOfVal name, heap' )
  where typeOfVal = typeOf value
        loc = freshLoc heap
        refs = refMap heap
        name = makeVarName Local loc
        heap' = heap { freshLoc = loc + 1,
                       refMap = (name, value) : refs }

Benefits

- Machine support: interpreter
- Proofs by hand: equational reasoning
- Automated testing and verification
Dependable DSLs: toward verified DSL compilers

FoF: a *semantics* language

**Compiler:** Convenient translator

**Interpreter:** DSL designer’s best support
Testing and validating

QuickCheck → Hamlet Semantics → valid/invalid

AST

Hamlet

FoF code

FoF Interpreter → FoF value → true/false

Big picture

• Hamlet input specifies the capability system
• E.g.: high-level retyping policy
• Enforce translation validity down to FoF code

Special bonus if your DSL is not Turing-complete
New perspective

Figure: Top-down approach

Figure: Bottom-up approach
Conclusion

High-level semantics language
- Give a *meaning* to DSLs
- Itself defined by a functional semantics
- For *provably correct* back-ends

Future work
- Translation validation [Necula, 2000]
- Optimization framework [Ramsey, 2010]
- More DSLs
1 Domain-specific languages for OS development
   - Why? Why not?
   - How?

2 Filet-o-Fish: practical DSL construction
   - An higher-order library
   - A compiled language

3 Filet-o-Fish: building dependable DSLs
   - Functional semantics
   - Toward verified DSL compilers
   - New perspective
Putting things together

Input

Parser  Sanity Checks

DSL Compiler

FoF Compiler

C Compiler

QuickCheck

AST  FoFC

DSL Semantics  FoF Semantics

FoF Interpreter

Filet-o-Fish

October 11, 2009 17 / 23
Filet-o-Fish correctness

\[ \text{FoFCode} \quad \xrightarrow{\text{Eval}} \quad \text{Trace} \quad \xrightarrow{\text{Compile}} \quad \text{CLight}^* \quad \xrightarrow{\text{Exec}} \quad \text{Trace} \quad \xrightarrow{\text{Simulation}} \]

*CLight: Mechanized semantics of C [Leroy, 2009]

Pen and Paper Proof

- On core + references
- But extremely tedious
- And quite involved
Figure: Workflow of the Hamlet compiler in FoF
Hamlet

Example of translation

/* Phys address range */
cap PhysAddr {
    retype_to {
        RAM: {base, bits},
        DevFrame: {base, bits},
        PhysAddr: {base, bits}
    }
    eq paddr base;
    eq uint8 bits;
}

bool is_well_founded(enum objtype fof_y1,
                     enum objtype fof_y2){
    switch (fof_y1) {
        case ObjType_PhysAddr: {
            return (((false
                          || (fof_y2 == ObjType_RAM))
                          || (fof_y2 == ObjType_DevFrame))
                          || (fof_y2 == ObjType_PhysAddr));
            break;
        }
    } ...
validateRetypeCode dstType (srcTypeV, validTypesP) =
  do
    return (srcTypeV, (do
        returnc $ cond validTypesP))
  where
    cond validTypes = foldl orType false validTypes
    orType x srcType = x .||. (dstType .==. srcType)

**Figure:** Snippet of the Hamlet compiler
Fugu
An Error Definition Language

errors retype_err {
    failure RETYPE_INVALID "Incorrect retype path",
    failure REVOKE_FIRST "Retyping an already retyped cap",
    success RETYPE_OK "Retyping success" 
};

Figure: Fugu Snippet

err_is_failF = def [Static, Inline] "err_is_fail"
    err_is_fail_int
    boolT [errval_tT]
where err_is_fail_int (err : []) =
    do
        err_no <- err_noF
        err_no_e <- call err_no [err]
        returnc (uint64 1 .-. (err_no_e .%. uint64 2))

Figure: Back-end Snippet (FoF code)
Tic-tac-toe retyping graph (2x2 board)