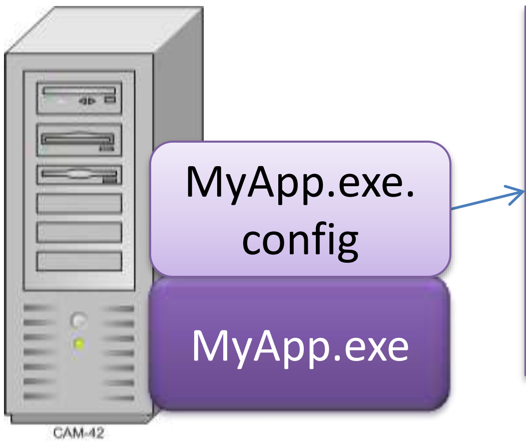
Semantic Subtyping with an SMT Solver

Cătălin Hrițcu, Saarland University, 16 December 2009

Joint work with Andy Gordon (MSR Cambridge),
Gavin Bierman (MSR Cambridge), and Dave Langworthy (MS Redmond)

The Oslo Modeling Language




```xml
<?xml version="1.0" encoding="utf-8"?>
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

- Server stacks (eg .NET) allow post-deployment configuration
  - But as server farms scale, manual configuration becomes problematic
  - Better to drive server configurations from a central repository
- M is a new modeling language for such configuration data
  - Ad hoc modeling languages remarkably successful in Unix/Linux world
  - M is in development (first “beta” Nov. 2008; most recent Nov. 2009)

Dynamic IT

The Problem

Planning Data
Requirements
KPIs, SLAs, etc.

Development Data
Architecture, Source Code, etc.

Little or no *data* sharing between tools/runtimes in the application lifecycle

Operation Data
Health, Policies, etc.
KPIs, SLAs, etc.

ISV Data
Rules, Process Models, etc.

Dynamic IT

Our Approach

Microsoft® Office

Microsoft® Visual Studio® Team System 2008

Microsoft® System Center™

IDS SCHEER Business Process Excellence

AMBERPOINT

inrule TECHNOLOGY

Planning Data

Requirements
KPIs, SLAs, etc.

Development Data

Architecture, Source Code, etc.

Operation Data

Health, Policies, etc.
KPIs, SLAs, etc.

ISV Data

Rules, Process Models, etc.

Tools/runtimes focus on experience/features (eg DSLs), data is shared in common models in SQL Server; M is language for typing and querying these models

Demo

- modules, functions, recursion (fact.m)
- types, entities, refinements (constraints.m)
- tagged unions, DSLs (WhileSimpler.m)
- collections, from-where-select, accumulate (types1.m and CauldronAccumulate.m)
- Types as predicates over values (typeful)
- Generating instances of types (inhabited)
  - Generating correct system configurations
  - Generating instances at runtime: enumerating multiple correct and incorrect system configurations

The Core of the M Language

- A **value** may be a **general value** (integer, text, boolean, null)
- Or a **collection** (an unordered list of values),
- Or an **entity** (a finite map from string labels to values)

- The expression

```
( from n in { 5, 4, 0, 9, 6, 7, 10}
  where n < 5
  select {Num=>n, Flag=>(n>0)} )
```

has the type

```
{Num:Integer; Flag:Logical;}*
```

and evaluates to

```
{{Num=>4,Flag=>true},
 {Num=>0, Flag=>false}}
```

Interdependent Types and Expressions

- A **refinement** type *T* where *e* consists of the values of type *T* such that boolean expression *e* holds
- A **typecase** expression *e* in *T* returns a boolean to indicate whether the value of *e* belongs to type *T*
  - {x=>1, y=>2} in {x:Any;} returns true (due to subtyping)
- A **type ascription** *e* : *T* requires that *e* have type *T*
  - Verify statically if possible
  - Compile to (*e* in *T*) ? *e* : throw "type error" if necessary

Primitive Types in D minor

- Named types (can be recursive) — X — type *X* : *T*;
- Top type — Any
- Scalar types — Integer32 — Text — Logical
- Collection types — { *T** }
- Entity types (at least field *l*) — { *l* : *T* }
- Refinement types (for a pure e) — (*T* where *e*)

Some Derived Types

- Empty type — Empty $\equiv$ Any where false
- Singleton type — {*e*} $\equiv$ Any where value==*e*
- Null type — Null $\equiv$ {null}
- Union type — *T* | *U* $\equiv$ Any where (value in *T* || value in *U*)
- Nullable type — Nullable *T* $\equiv$ *T* | {null}

Some More Derived Types

- Intersection type

  $T\ \&\ U \equiv$ Any where (value in T && value in U)

- Negation type

  $!T \equiv$ Any where !(value in T)

- Multi-field entity type

  $\{f_1{:}T_1; f_2{:}T_2\} \equiv \{f_1{:}T_1\}\ \&\ \{f_2{:}T_2\}$

- Closed entity type (enforce eta)

  closed $\{f_1{:}T_1; f_2{:}T_2\} \equiv \{f_1{:}T_1; f_2{:}T_2\}$ where value == $\{f_1 \Rightarrow \text{value}.f_1, f_2 \Rightarrow \text{value}.f_2\}$

- Self type

  Self(value)$U \equiv$ Any where (value in U)

Type-checking

- Type assignment relation ($E \vdash e : T$)
  - if $E \vdash e : \{l : T\}$ then $\Gamma \vdash e.l : T$ (field selection)
  - if $E \vdash e : T$ and $E \vdash T <: U$ then $E \vdash e : U$ (subsumption)
  - if $E \vdash e : T$ and e pure then $E \vdash e : \mathrm{T}$ where value == e (singleton)
  - This is just a specification of what a type-checker should do
- Type-checking algorithm by "bidirectional rules" (as e.g. in C#)
  - $\mathrm{E} \vdash \mathrm{e} \rightarrow \mathrm{T}$ (type synthesis) and $\mathrm{E} \vdash \mathrm{e} \leftarrow \mathrm{T}$ (type checking)
- Subtyping decided semantically, by external SMT prover
  - $E \vdash T <: U$ when Axioms $\models$ F[| E |] => F[| T |](x) => F[| U |](x)

Purity

- D minor side-effects: non-termination and non-determinism
- The *e* in the type *(T where e)* has to be "pure"
  - Pure expressions have a (unique) normal form
- Checking expression purity:
  - $f(e_1, \ldots, e_n)$ should terminate ("bad" uses of recursion disallowed)
  - *e* in *T* (and *e : T*) should terminate even when T is recursive (recursive types used with "in" need to be "contractive")
  - from *x* in e_1 let y = e_2 accumulate e_3 should converge ("$\lambda x\ y.\ e_3$" needs to be associative and commutative)

First-order theories

- Semantics given with respect to a particular logical model
- We use SMT-LIB (+Z3 extensions) to axiomatize this model
- Sorted first-order logic +
  - Integers: build-in sort Int + arithmetic operations

```
:formula (forall (x Int) (= (+ 0 x) x))      ; Z3: valid
```

  - Algebraic datatypes:

```
:datatypes((VList
     Nil
     (Cons (out_Head Value) (out_Tail VList))))
```

  - "Arrays" – updatable functions with finite support

```
:define_sorts ((VArray (array Int Value))     ; C arrays
      (VBag (array Value Int))           ; M collections
      (VMap (array String Value)))          ; M entities
```

Logical model

- The semantic domain of values

```
:datatypes (
  (Value
    (G (out_G General))                   ;; scalar values
    (E (out_E (array String Value)) ;; entities
    (C (out_C (array Value Int)))   ;; collections
    (L (out_L VList)))                    ;; lists
  (VList Nil (Cons (out_Head Value) (out_Tail VList))))
```

- Axiomatization of function and predicate symbols

```
:extrafuns((v_tt Value)(v_int Int Value)(O_Sum Value Value Value))
:assumption (= v_tt (G(G_Logical true)))
:assumption (forall (n Int) (= (v_int n) (G(G_Integer n)))
  :pat { (v_int n) } :pat { (G(G_Integer n)) }
:assumption (forall (i1 Int) (i2 Int)
  (= (O_Sum (v_int i1) (v_int i2)) (v_int (+ i1 i2)))
  :pat { (O_Sum (v_int i1) (v_int i2)) })
```

Axiomatizing collections

- Finiteness of bags

```
:assumption (forall (a (array Value Int))
  (iff (Finite a) (= (default a) 0)))
```

- Only positive indices in bags

```
:assumption (forall (a (array Value Int))
  (iff (Positive a) (forall (v Value) (>= (select a v) 0))
```

- Collections are finite bags with positive indices

```
:assumption (forall (v Value)
  (iff (In_C v)
       (and (is_C v)
            (Finite (out_C v))
            (Positive (out_C v)))))
```

- Collection membership

```
:assumption (forall (v Value) (a (array Value Int))
  (iff (v_mem v (C a)) (> (select a v) 0)))
```

Semantics

- Semantics of types:
  F[| *T* |](*x*) is a FOL formula where *x* ranges over sort Value
  - F[| Any |](*x*) = true
  - F[| { *T** } |](*x*) = In_C(*x*) ∧ (forall (*y* Value) v_mem *y* *x* => F[| *T* |](*y*))
  - F[| *T* where *e* |](*x*) = F[| *T* |](*x*) ∧ let value = *x* in [| *e* |] = v_tt ...
  - Logical soundness: If *E* ⊢ *e* : *T* then F[| *E* |] => F[| *T* |]([| *e* |])
- Semantics of pure expressions: [| *e* |] is a FOL term
  - [| e_1 + e_2 |] = O_Sum [| e_1 |] [| e_2 |]
  - [| *e* in *T* |] = if F[| *T* |]([| *e* |]) then v_tt else v_ff ...
  - Full abstraction: If *e*, *e'* are pure then
    e →* *v* *← *e'* iff [| *e* |] = [| *e'* |]; in particular *e* →* *v* iff [| *e* |] = *v*

Microsoft Research Cambridge

THE END