Reduction and typing of complex classes

Peter Vanderbilt
pvanderbilt@acm.org

Introduction

- Interest in mutually-recursive nested classes
    - Language features
    - Statically type correct (no runtime casts)
- Reductions: mapping high-level constructs to lower-level terms
    - Provide semantics
    - Validate, or derive, type rules
- Nested classes lead to virtual types, “complex content” and “hybrids”
- The reduction of this paper handles complex content, a step towards nested classes

Very simple reduction

- Class definition
  - top-level, closed, simple content
  - “self” (aka “this”) denotes the future object and is captured by λ-expressions in e*
  - result: binds C and “**new** C”
- Reduction
  - reduces to type and function
  - **new** C ≜ C_new()

class *C* { **const** *x** : *T** = *e** } ⟼
  type *C* = { *x** : *T** };
  func *C*_new () = μ(self) { *x** = *e** }

```
class A {
   const x : Int = 5
   const get_x : ()-->Int = λ()self.x
} ⟼
   type A = { x : Int; gx : ()-->Int };
   func A_new () =
      μ(self) { x= 5; get_x = λ()self.x }
```

Type-preservation

- A type-correct program will be type correct when reduced
  - Prove that the reduction is type correct given premises and yields the right declarations
  - And has correct properties
    - "**new** C" : *C*
- Actually derive high-level rule from reduction
  - What premises are needed for a subproof of the reduction
  - Explore restrictions & extensions

class *C* { **const** x^* : T^* = e^* } ⟼
type *C* = { x^* : T^* };
func *C*_new () = μ(self) { $x^* = e^*$ }

type *C*={x^*:T^*}; **func** *C*_new()=μ(self){x^*=e^*}

$$\frac{\Gamma \vdash T^* :: *}{\Gamma \vdash \textbf{class}\ C\ \{\ \textbf{const}\ x^* : T^* = e^*\ \} \rightsquigarrow C\texttt{<-->}\{x^*:T^*\};\ C\_new : ()\texttt{-->}C}$$

Nesting

- Goal: reduce nested class definitions by nested reductions:
  - reduce inner, then
  - reduce outer
- But inner reduction has a type definition (and other issues)
  - class B has “complex” content
  - b:B => b.A is a type, “new b.A” (aka b.A_new()) creates instances
  - b is a “hybrid”

```
class B {
    class A { const x : Int  = 5 };
    const f ()-->A = λ() new A; ...
}
⟼ // reduce inner
    class B {
        type A = { x : Int };
        func A_new () : A = μ(self) { x = 5 };
        const f : ()-->A = λ()A_new(); ...
    }
⟼ // reduce outer -- oops!
```

Hybrids

- Hybrid expressions (*h*)
  - Hybrid literal: { ... }
    - **const** *x* = *e*
    - **meta** *X* = *M*
    - **hconst** *x* = *h*
  - also λ, (), μ, **let-in**, **with**, ...
- Hybrid types (*H*)
  - type literals: { ... }
    - **const** *x* : *T*
    - **meta** *X* : *Z*
    - **hconst** *x* : *H*
  - τ[*r*] *H*
  - also ...

Z ::= {{?}} | {{?⊙*T*}} | ...
⊙∈{↔,<:,:>}

- h = { **type** T = Int; **const** x = 5 }
  - h.x is an Int
  - h.T is a type expression (=Int)
  - "**const** c : h.T = h.x" is OK

type T : {{?↔Int}}

- h : { **type** T ↔ Int; **const** x : Int }

-[r] { **type** T; **const** x : r.T }

Definitions of classes with complex content

J^C :: = **class** *C* **extends** *D* { F^* }

F ::= [**member** | **static**] J

J ::= [**virtual** | **refine**] V
| **abstract** *L*
| (**impl** | **override**) *F*

V ::= **const** *x* : *T* = *e*
| **meta** *X* :: *K* : *Z* = *M*
| **hconst** *x* : *Tg* = (*Te*) *e*

Some abbreviations:

func *f* (*x**:*T**) : *V* = *e* ≈
const *f* : (*T**)-->*V* = λ(*x**)*e*

type *X* : *Z* = *T* ≈ **meta** *X* :: * : *Z* = *T*

type +*X* = *T* ≈ **meta** *X* :: * : {{?<:*T*}} = *T*

Result of class definitions

- After "**class** *C* ...", several entities exist:
  - "**new** *C*": creates instances of the class -- which have member fields
  - *C*@ and *C*# are types of instances
  - *C* is a "class object," which has static fields and more
  - {{?<<*C*}} is a type with class objects as its instances
  - "**extends** *C*" is available for extending (subclassing) *C*
- Partial classes do not have: **new** *C*, *C*@ or *C*.

Types *C@* and *C#*

- Type *C@* is the “exact” type
  - Instances of *C@* are those of the class (but not subclasses)
    - *C@* provides type-safe access to all fields
- Type *C#* is the “general” (or “hash”) type
  - Instances of *C#* are those of this class and its subclasses
- *C#* provides for subclass polymorphism
- But for variables of type *C#*, fields whose types reference virtual types require special handling
- **new** *C* : *C@* <: *C#* <: *D#* (for superclass *D*)
  - *C@* & *D@* are incompatible

Class objects

- A class object named *C* is one thing produced by a class definition for concrete *C*.
- Class object *C* is a hybrid containing:
  - Static fields (*C*.*s*)
  - type “Instance” of instances of the class
  - function new : ()-->*C*.Instance
- Where
  - *C*@ ≜ *C*.instance
  - “**new** *C*” ≜ *C*.new()
- *C* is a first-class object
- {{?<<*C*}} is a type containing *C* and class objects of subclasses of *C*
  - *C* : {{?<<*C*}} ≜ *C*<<*C*
  - For any subclass, *B*, of *C*, *B*<<*C* (≜B:{{?<<*C*}})

Special variables

- “self” (aka “this”) refers to the future instance object
- “Self” (cap “S”) refers to the future class object
- “*x* = **new** *C*” implies self=*x* and Self=*C*
- They can appear in implementations
- They can also appear in types
- They provide access to virtual types (in canonical form)
  - self.*T* is member virtual type *T*
  - Self.*T* is static virtual type *T*
    - Self@ is the “self-type”
    - self : Self@
- Also “super” and “Super” can appear in implementations (but not types)

Basic form of the reduction

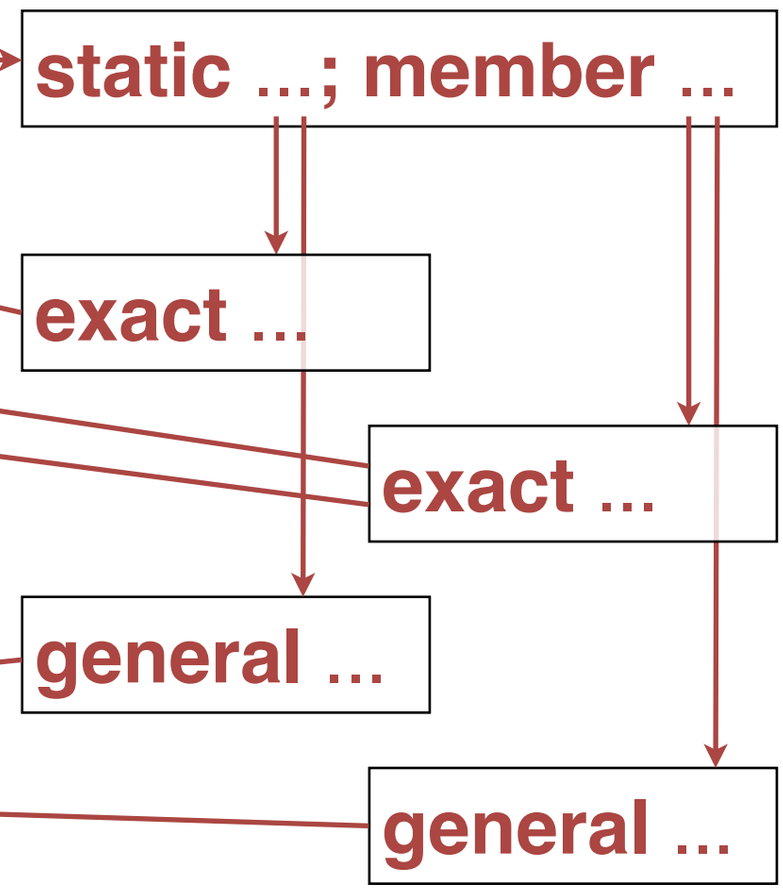
class *C* { ... } ⟼
  hconst *C* = μ(Self) {
    /* definitions of static fields */
    type Instance = {...};
    func new () : Self@ = μ(self) {...};
  };
  htype *C*_gT = τ[Self] {
    /* declarations of static fields */
    type Instance <: {...};
  };
  htype *C*_hT = ∪[Self<<*C*] Self@
    = ∪[Self:*C*_gT] Self.Instance
  ...

new *C*	≜	*C*.new()
C@	≜	*C*.Instance
C#	≜	*C*_hT
{{ ?<<*C* }}	≜	*C*_gT
C.s	≜	*C.s*

B << *C*	≜	*B* : {{? << *C*}}
	≜	*B* : *C*_gT

Rewriting the class body

```
class C { ... } ⟼
   hconst C = μ(Self) {
      /* definitions of static fields */
      type Instance = {...};
      func new () : Self@ = μ(self) {...};
   };
   htype C_gT = τ[Self] {
      /* declarations of static fields */
      type Instance <: {...};
   };
   htype C_hT = ∪[Self<<C] Self@
       = ∪[Self:C_gT] Self.Instance
   ...
```



Exact & general aspects

class $C\{\, F^* \,\}$

$F^* \approx$ **static** J^*_s ; **member** J^*_m

$J^*_s \approx$ **exact** $x^*_{se} :: K^*_{se} : T^*_{se} = e^*_s$; **general** $x^*_{sg} :: K^*_{sg} : T^*_{sg}$

$J^*_m \approx$ **exact** $x^*_{me} :: K^*_{me} : T^*_{me} = e^*_m$; **general** $x^*_{mg} :: K^*_{mg} : T^*_{mg}$

$\lambda()\mu(self)\{\, x^*_{me} = e^*_m \,\} : \tau[self]\{\, x^*_{me} : T^*_{me} \,\} <: \tau[self]\{\, x^*_{mg} : T^*_{mg} \,\}$

new *C* : *C@* <: *C#*

virtual & **refine**	≈	exact & general
abstract	≈	just general
impl & **override**	≈	just exact

Rewrites

const *x* : *T* = *e*	≈	**exact** *x* :: ε : *T* = *e* ; **general** *x* :: ε : *T*
meta *X* :: *K* : *Z* = *M*	≈	**exact** *X* :: *K* : {{?<-->*M*}} = *M* ; **general** *X* :: *K* : *Z*
hconst *x* :: *K* : *Tg* = (*Te*) *e*	≈	**exact** *x* :: *K* : *Te* = *e* ; **general** *x* :: *K* : *Tg*
type *X* : *Z* = *T*	≈	**exact** *X* :: * : {{?<-->*T*}} = *T* ; **general** *X* :: * : *Z*
type +*X* = *T*	≈	**exact** *X* :: * : {{?<-->*T*}} = *T* ; **general** *X* :: * : *Z*

Summary of the Reduction

generic (inheritable) part

class *C* **extends** *D* { ... } ⟼

htype *C*_αβTc [...] = *D*_αβTc[...] **with** { $x^*_{\alpha\beta}$: $T^*_{\alpha\beta}$ };

hfunc *C*_αc (...) : *C*_αβTc[...] =
  let super = *D*_αc(...) **in** super **with** { $x^*_{\alpha\beta} = e^*_{\alpha}$ };

α ∈ {m,s}
β ∈ {e,g}

class-specific part

hconst *C* = μ(*C*) *C*_sc(*C*) **with** {
  type Instance = τ[self] *C*_meTcc[*C*,self];
  func new () : Self@ = μ(self) *C*_mc(*C*,self);
};

htype *C*_gT = τ[Self] *C*_sgTc[Self] **with** {
  type Instance <: τ[self] *C*_mgTcc[Self,self];
};

htype *C*_hT = ∪[Self<<*C*] Self@;

Typing method implementations

hfunc *C*_mc (*Self:*C*_gT,*self:Self@) : *C*_meTcc[Self,self] =
    let super = *D*_mc(Self,self) **in** super **with** { x^*_{me} = e^*_{m} };

Γ; ...; Self:*C*_gT; self:Self@; super:*D*_meTcc[Self,self]
    ⊢ e^*_{m} : T^*_{me}

note:
    self : *C*_mgTcc[Self,self] = *C*_mgTcc[Self,self] **with** { x^*_{mg} : T^*_{mg} }
    Self : *C*_sgTc[Self] = *C*_sgTc[Self] **with** { x^*_{sg} : T^*_{sg} }

Typing calls

p : C@ ==>

p : C_meTcc[C,p] <: { x^*_{me} : T^*_{me}<Self↦C, self↦p> }
p.Class <--> C

p : C# ==>

p : C_mgTcc[p.Class,p] <: { x^*_{me} : T^*_{me}<Self↦p.Class, self↦p> }
p.Class << C

Example: Class object as parameter

```
interface Buildable [T] {
    static func new () : Self@
    func add (T) : Void
}
```

⟼

```
htype Buildable_gT [T] = τ[Self] {
    type Instance <: { func add (T) : Void }
    func new () : Self@
}
```

```
partial class Iterable [T] {
    ...
    func mapTo [R] (f:(T)→R, *C << Buildable[R]) : C@ {
        const c = new C;
        foreach (e in self.elements()) { c.add(f(e)) }
        return c
    } ... }
```

```
C << Buildable[R] ≜
C : Buildable_gT[R] ==>
    C@ <: { add : (R)-->Void }
    C.new : ()-->C@
```

```
c.mapTo[Int](f,IntSet) : IntSet@
    given c : Iterable[String]#  and  f : (String)-->Int and IntSet<<Buildable[Int]
```

Example: Hybrid field -- in implementation & type

```
partial class Iterable [T] {

   func mapTo [R] (f:(T)→R,
      *C << Buildable[R]) : C@ = ...;

   static abstract hconst
        DefC[X] << Buildable[X];

   func map [R] (f:(T)→R)
       : Self.DefC[R]@
       = self.mapTo[R](f,Self.DefC[R]);
   ... }
```

```
Γ; T :: *; Self : Iterable_gT[T];
      self : Self@; R :: *; f : (T)-->R
  ⊢ self.mapTo[R](f,Self.DefC[R])
            : Self.DefC[R]@
```

⟼

```
htype Iterable_gT [T] = τ[Self] {
    DefC[X] << Buildable[X];
    Instance <: {
       mapTo: [R]-->(...)-->C@;
       map: [R]-->(...)-->Self.DefC[R]@
     }
}

func Iterable_mc [T]
   (Self<<Iterable[T],self:Self@) =
{
    mapTo[R](f,C) = ....;
    map[R](f) = self.mapTo...;
}
```

Example: Handling “Self” in return type

```
class List [T]
   extends Buildable[T], Iterable[T]
{
    static impl hconst DefC = List;
    ...
}
class Set [T]
   extends Buildable[T], Iterable[T]
{
    static impl hconst DefC = Set;
    ...
}
class IntSet extends Set[Int] { ... }
```

```
List_meTc [T] [Self] <--> { ...;
   map: ... -->Self.DefC[R]@
}
List [T] : { ...;
    hconst DefC <--> List;
    type Instance <--> List_meTc[List[T]]
}
```

```
c : List[T]@ ==> c.map(f) : List[R]@
c : Set[T]@ ==> c.map(f) : Set[R]@
c : IntSet@ ==> c.map(f) : Set[R]@
```

```
c : Iterable[T]#  ==>
   c.map(f) : U[Self<<Buildable[R]]
        Self.DefC[R]@
   c.map(f) : self.Class.DefC[R]@
   c.map(f) : Buildable[R]#
```

Summary: What this work is about

- Classes with “complex content”: data, types (and higher-order ones), nested hybrids
    - Types of fields depend on *values* of other type fields (self.*m*, Self.*s*)
- Reduction of complex class definitions to hybrid types functions & objects
    - Hybrids are objects with complex content (language and logic to come)
    - Includes a class object with Instance (*C*@) and new (**new** C)
- Typing rule that follows from the reduction

Future research

- Publish the lower-level language (of hybrids) and its logic
- Define reductions that support “sibling” or “current value” reference
- Define reductions that yield one object with everything inside

Thank you. Questions?

pvanderbilt@acm.org
petervanderbilt.com/research