Dummy title

Authors omitted for double-bind review.
Unspecified Institution.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Formal

$$
\begin{array}{rcll}
id & ::= & t \mid C & \\
T & ::= & \texttt{This}\,n.\,Cs & \\
CD & ::= & C\texttt{=}E & \text{class declaration} \\
CV & ::= & C\texttt{=}LV & \text{evaluated class declaration} \\
D & ::= & id\texttt{=}E & \text{declaration} \\
DL & ::= & id\texttt{=}L & \text{partially-evaluated-declaration} \\
DV & ::= & id\texttt{=}LV & \text{evaluated-declaration} \\
L & ::= & \texttt{interface}\,\texttt{\{}Tz\texttt{;}\ amtz\ \texttt{;}\ \texttt{\}} \mid \texttt{\{}Tz\texttt{;}\ Ms\ \texttt{;}\ K?\texttt{\}} & \text{literal} \\
LV & ::= & \texttt{interface}\,\texttt{\{}Tz\texttt{;}\ amtz\ \texttt{;}\ \texttt{\}} \mid \texttt{\{}Tz\texttt{;}\ MVs\ \texttt{;}\ K?\texttt{\}} & \text{literal value} \\
amt & ::= & T\ m\texttt{(}Txs\texttt{)} & \text{abstract method} \\
mt & ::= & T\ m\texttt{(}Txs\texttt{)}\ e? & \text{method} \\
Tx & ::= & T\ x & \text{paramater-declaration} \\
M & ::= & CD \mid mt & \text{member} \\
MV & ::= & CV \mid mt & \\
Mid & ::= & C \mid m & \text{member-id} \\
K & ::= & \texttt{constructor}\texttt{(}Txs\texttt{)} & \text{constructor} \\
e & ::= & x \mid e.\,m\texttt{(}es\texttt{)} \mid e.\,x \mid \texttt{new}\ T\texttt{(}es\texttt{)} & \text{expression} \\
E & ::= & L \mid t \mid E \texttt{ <+ } E \mid E\texttt{(}Cs\texttt{=}T\texttt{)} & \text{library-expression} \\
\mathcal{E}_V & ::= & \square \mid \mathcal{E}_V \texttt{ <+ } E \mid LV \texttt{ <+ } \mathcal{E}_V \mid \mathcal{E}_V\texttt{(}Cs\texttt{=}T\texttt{)} & \text{context of library-evaluation} \\
\mathcal{E}_v & ::= & \square \mid \mathcal{E}_v.\,m\texttt{(}es\texttt{)} \mid v.\,m\texttt{(}vs\ \mathcal{E}_v\ es\texttt{)} \mid \mathcal{E}_v.\,x \mid \texttt{new}\ T\texttt{(}vs\ \mathcal{E}_v\ es\texttt{)} & \\
v & ::= & \texttt{new}\ T\texttt{(}vs\texttt{)} & \\
p & ::= & DLs\texttt{;}\ DVs & \text{program} \\
S & ::= & Ds\ e & \text{source code}
\end{array}
$$

We use t and C to syntactically distinguish between trait and class names. A type (T) has an interesting syntax, see below for what it means. An E is a top-level class expression, which can contain class-literals, references to traits, and operations on them, namely our sum $E < +E$ and redirect $e(Cs = T)$. A declaration D is just an $id = E$, representing that id is

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:6
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

declared to be the value of E, we also have CD, CV, DL, and DV that constrain the forms of the LHS and RHS of the declaration. A literal L has 4 components, an optional interface keyword, a list of implemented interfaces, a list of members, and an optional constructor. For simplicity, interfaces can only contain abstract-methods (amt) as members, and cannot have constructors. A member M, is either an (potentially abstract) methood mt or a nested class declaration (CD). A member value MV, is a member that has been fully compiled. An mid is an identifier, identifying a member. Constructors, K, contain a Txs indicating the type and names of fields. An e is normal fetherweight-java style expression, it has variables x, method calls $e.m(es)$, field accesses $e.x$ and object creation $newes$. $CtxV$ is the evalation context for class-expressions E, and $ctxv$ is the usuall one for e's.

An S represents what the top-level source-code form of our language is, it's just a sequence of declarations and a main expression. The most interesting form of the grammer is a p, it is a 'program', used as the context for many reductions and typing rules, on the LHS of the ; is a stack representing which (nested) declaration is currently being processed, the bottom (rightmost) DL represents the D of the source-program that is currently being processed. Th RHS of the ; represents the top-level declarations that have allready been compiled, this is neccessary to look up top-level classes and traits.

To look up the value of a type in the program we will use the notation $p(T)$, which is defined by the following, but only if the RHS denotes an LV:

$$\begin{aligned}(;\_, C = L, \_)(\texttt{This}0.C.Cs) &\coloneqq L(Cs)\\ (id = L, p)(\texttt{This}0.Cs) &\coloneqq L(Cs)\\ (id = L, p)(\texttt{This}n+1.Cs) &\coloneqq p(\texttt{This}n.Cs)\end{aligned}$$

To get the relative value of a trait, we define $p[t]$:

$$(DLs;\_, t = LV, \_)[t] \coloneqq LV[\texttt{This}\#DLs]$$

To get a the value of a literal, in a way that can be understand from the current location ($\texttt{This}0$), we define:

$$p[T] \coloneqq p(T)[T]$$

And a few simple auxiliary definitions:

$$\begin{aligned}Ts \in p &\coloneqq \forall T \in Ts \bullet p(T) \text{ is defined}\\ L(\emptyset) &\coloneqq L\\ L(C.Cs) &\coloneqq L(Cs) \text{ where } L = \texttt{interface}?\,\{\_;\ \_, C = L, \_;\ \_\}\\ L[C = E'] &\coloneqq \texttt{interface}?\,\{Tz;\ MVs\ C = E'\ Ms;\ K?\}\\ &\quad \text{where L} = \texttt{interface}?\,\{\text{Tz};\ MVs\ \text{C} = \_\ Ms\ ;\ \text{K?}\}\end{aligned}$$

We have two-top level reduction rules defining our language, of the form $Dse^{\smile\smile} > Ds'e$ which simply reduces the source-code. The first rule (*compile*) 'compiles' each top-level declaration (using a well-typed subset of allready compiled top-level declarations), this reduces the defining expresion. The second rule, (*main*) is executed once all the top-level declarations have compiled (i.e. are now fully evaluated class literals), it typechecks the top-level declarations and the main expression, and then procedes to reduce it. In principle only one-typechecking is needed, but we repeat it to avoid declaring more rules.

```
Define Ds e --> Ds' e'
==================================================================
DVs' |- Ok
empty; DVs'; id | E --> E'
(compile)----------------------------------------- DVs' subsetof DVs
DVs id = E Ds e --> DVs id = E' Ds e

DVs |- Ok
DVs |- e : T
DVs |- e --> e'
(main)---------------------------------- for some type T
DVs e --> DVs e'
```

2 Compilation

Aside from the redirect operation itself, compilation is the most interesting part, it is defined by a reduction arrow $p; id| - E - - > E'$, the id represents the id of the type/trait that we are currently compiling, it is needed since it will be the name of $This0$, and we use that fact that that is equal to $This1.id$ to compare types for equality. The $(CtxV)$ rule is the standard context, the (L) rule propegates compilation inside of nested-classes, $(trait)$ merely evaluates a trait reference to it's defined body, (sum) and $(redirect)$ perform our two meta-operations.

```
Define p; id |- E --> E'
===========================================================
p; id |- E --> E'
(CtxV) ----------------------------------------------
p; id |- CtxV[E] --> CtxV[E']

id = L[C = E], p; C |- E --> E'
(L) --------------------------------------------- // TODO use fresh C?
p; id |- L[C = E] ---> L[C = E']

(trait) -----------------------------------
p; id |- t -> p[t]

LV1 <+p' LV2 = LV3                 p' = C' = LV3, p
(sum) --------------------------------------- for fresh C'
p; id |- LV1 <+ LV2 --> LV3

// TODO: Inline and de-42 redirect formalism
(redirect) ------------------------------------LV'=redirect(p, LV, Cs, P)
p; id |- LV(Cs=P) -> LV'
```

3 The Sum operation

The sum operation is defined by the rule $L1 < +pL2 = L3$, it is unconventional as it assumes we allready have the result ($L3$), and simply checks that it is indead correct. We believe (but have not proved) that this rule is unambigouse, if $L1 < +pL2 = L3$ and $L1 < +pL2 = L3'$, then $L3 = L3'$ (since the order of members does not matter for Ls).

The main rule fir summong of non-interfaces, sums the members, unions the implemented interfaces (and uses *mininize* to remove any duplicates), it also ensures that at most one of them has a constructor. For summing an interface with a interface/class we require that an interface cannot 'gain' members due to a sum. The actually L42 implementation is far less restrictive, but requires complicated rules to ensure soudness, due to problems that could arise if a summed nested-interface is implemented. Summing of traits/classes with state is a non-trivial problem and not the focus of our paper, their are many prior works on this topic, and our full L42 language simply uses ordinary methods to represent state, however this would take too much effort to explain here.

```
Define L1 <+p L2 = L3
===========================================================================================
{Tz1; Mz1; K?1} <+p {Tz2; Mz2; K?2} = {Tz; Mz; K?}
Tz = p.minimize(Tz1 U Tz2)
Mz1 <+p Mz1 = Mz
{empty, K?1, K?2} = {empty, K?} //may be too sophisticated?

interface{Tz1; amtz,amtz';} <+p interface?{Tz2;amtz;} = interface {Tz;amtz,amtz';}
Tz = p.minimize(Tz1 U Tz2)
if interface? = interface then amtz'=empty
```

The rules for summing member are simple, we take two sets of members collect all the oness with unique names, and sum those with duplicates. To sum nested classes we merely sum their bodies, to sum two methods we require their signatures to be identical, if they both have bodies, the result has the body of the RHS, otherwise the result has the body (if present) of the LHS.

```
Define Mz <+p Mz' = Mz"
-----------------------------------------
M, Mz <+p M', Mz' = M <+p M', Mz <+p Mz
//note: only defined when M.Mid = M'.Mid

Mz <+p Mz' = Mz, Mz':
dom(Mz) disjoint dom(Mz')

Define M <+p M' = M"
----------------------------------------
T' m(Txs') e? <+p T m(Txs) e = T m(Txs) e
T', Txs'.Ts =p Ts, Txs

T' m(Txs') e? <+p T m(Txs) = T m(Txs) e?
T', Txs'.Ts =p Ts, Txs

(C = L) <+p (C = L') = L <+p.push(C) L'
```

4 Type System

The type system is split into two parts: type checking programs and class literals, and the typechecking of expressions. The latter part is mostly convential, it involves typing judgments of the form $p; Txs \vdash e : T$, with the usual program p and variable environement Txs (often called Γ in the literature). rule ($Dsok$) type checks a sequence of top-level declarations by simply push each declaration onto a program and typecheck the resulting program. Rule *pok* typechecks a program by check the topmost class literal: we type check each of it's members (including all nested classes), check that it properly implements each interface it claims to, does something weird, and finanly check check that it's constructor only referenced existing types,

```
Define p |- Ok
============================================================

D1; Ds |- Ok ... Dn; Ds|- Ok
(Ds ok) ------------------------------ Ds = D1 ... Dn
Ds |- Ok

p |- M1 : Ok .... p |- Mn : Ok
p |- P1 : Implemented .... p |- Pn : Implemented
p |- implements(Pz; Ms) /*WTF?*/                  if K? = K: p.exists(K.Txs.Ts)
(p ok) ------------------------------------------- p.top() = interface? {P1...Pn; M1, ..., Mn; K?
p |- Ok

p.minimize(Pz) subseteq p.minimize(p.top().Pz)
amt1 _ in p.top().Ms ... amtn _ in p.top().Ms
(P implemented) ------------------------------------------------ p[P] = interface {Pz; amt1 ... am
p |- P : Implemented

(amt-ok) ------------------- p.exists(T, Txs.Ts)
p |- T m(Tcs) : Ok

p; This0 this, Txs |- e : T
(mt-ok) ------------------------------ p.exists(T, Txs.Ts)
p |- T m(Tcs) e : Ok

C = L, p |- Ok
(cd-Ok) -------------------
p |- C = L : OK
```

Rule ($Pimplemented$) checks that an interface is properly implemented by the program-top, we simply check that it declares that it implements every one of the interfaces super-interfaces and methods. Rules ($amt - ok$) and ($mt - ok$) are straightforward, they both check that types mensioned in the method signature exist, and ofcourse for the latter case, that the body respects this signature.

To typecheck a nested class declaration, we simply push it onto the program and typecheck the top-of the program as before.

The expression typesystem is mostly straightforward and similar to feartherwieght Java, notable we we use $p[T]$ to look up information about types, as it properly 'from's paths, and use a classes constructor definitions to determine the types of fields.

```
Define p; Txs |- e : T
====================================
(var)
----------------------- T x in Txs
p;  Txs |- x : T

(call)
p; Txs |- e0 : T0
...
p; Txs |- en : Tn
-----------------------------------  T' m(T1 x1 ... Tn xn) _ in p[T0].Ms
p; Txs |- e0.m(e1 ... en) : T'

(field)
p; Txs |- e : T
---------------------------------------  p[T].K = constructor(_ T' x _)
p; Txs |- e.x : T'


(new)
p; Txs |- e1 : T1 ... p; Txs |- en : Tn
------------------------------------------- p[T].K = constructor(T1 x1 ... Tn xn)
p; Txs |- new T(e1 ... en)


(sub)
p; Txs |- e : T
-----------------------------------  T' in p[T].Pz
p; Txs |- e : T'


(equiv)
p; Txs |- e : T
-----------------------------------  T =p T'
p; Txs |- e : T'
```

– towel1:.. //Map: towel2:.. //Map: lib: T:towel1 f1 ... fn MyProgram: T:towel2 Lib:lib[.T=This0.T] ... –

References