cardelli's linker

Daniel Jackson
MIT Lab for Computer Science
6898: Advanced Topics in Software Design
March 11, 2002

cardelli's motivation

separate compilation
- vital for writing, delivering, maintaining libraries
- but often things go wrong
- module designers lost sight of original aims

linking has become complicated
- so develop a formal model to help reason
- treat linking outside the programming language

Luca Cardelli. Program Fragments, Linking & Modularization. POPL 1997

separate compilation goofs

missing language features

- no interface/impl distinction, esp. in untyped languages

global analysis required

- multimethods, overloading, ML modules

can't compile library without client

- templates in C++, Ada, Modula-3
- overloading in Java (David Griswold)

runtime type errors despite static types

- covariance in Eiffel
- may link with old library by mistake
- class loading problems

our motivation

Units: a new modularity mechanism
- will be presented on wednesday by Findler
- have much in common with Cardelli's model

nice example of simple theory
- shows how to capture essence of a problem
- use as reference model for more complex systems

challenge: a nice final project
- recast Cardelli's theory in Alloy
- might be simpler: no decl ordering
- check theorems automatically
- explore variants

key ideas

focus on type checking: the hardest part

- work at source code level
- compilation is fragmenting bindings
- linking is substitution (ie, inlining)

linksets: a simple configuration language

- a linkset is a collection of fragments to be linked
- each fragment has
  - a name
  - a list of imports
  - an export
- a linkset has an external interface
  - its imports and exports
  - empty for a program, non-empty for a library

judgments

the judgment $E \vdash a: A$ means

- in the environment E
- expression a has type A

examples

- $f: int \rightarrow int \vdash f(3): int$
- $f: int \rightarrow int, i: int \vdash f(i): int$
- $g: int \rightarrow int \vdash f(x)\{return\ g(x)\}: int \rightarrow int$

environment

- $\emptyset, x_1 : A_1, \ldots, x_n : A_n$
- well formed if x_i != x_j for i!= j

lambda calculus

lambda calculus

- a toy language for theoretical investigation
- if you understand Scheme, you'll understand it
- Cardelli uses the variant 'F1': explicit, first-order types

syntax

- types

A, B ::= K	base type
| A→B	function type

- terms

a, b ::= x	variable
| lambda (x: A) b	abstraction
| b (a)	application

type checking

expressed as inference rules

- if f has the type $A \rightarrow B$ (in environment E)
- … and a has the type A
- … then f(a) has the type B

$$\frac{E \vdash f : A \rightarrow B, \quad E \vdash a : A}{E \vdash f(a) : B}$$

$$\frac{E, x : A \vdash b : B, \qquad E \vdash a : A}{E \vdash \text{lambda } (x : A)\ b : A \rightarrow B}$$

$$\frac{E \ \vdash \ \diamond}{E \vdash x : E(x)}$$

linksets

$\emptyset |$
$f \# (\emptyset \vdash \text{lambda (x: int) x}: int \rightarrow int),$
$main \# (\emptyset, f: int \rightarrow int \ \vdash f(3) : int)$

represents these two fragments:

```
fragment f: int -> int
import nothing
begin lambda (x: int) x end
```

```
fragment main: int
import f: int -> int
begin f(3) end
```

definitions

general form of a linkset

- $L \equiv E_0 \mid x_1 \# \; E_1 \vdash \; a_1 : A_1 , \ldots, x_n \# \; E_n \vdash \; a_n : A_n$

defined notions

- imported names $imp(L) = dom\ (E_0)$
- exported names $exp(L) = \{x_1, \ldots, x_n\}$
- imported environments $imports(L) = E_0$
- exported environments $exports(L) = \{x_1 : A_1, \ldots, x_n : A_n\}$

well formed iff

- imports(L) and exports(L) are environments
- $(E_0, E_i)$is an environment
- $dom\ E_i \subseteq \; exp(L)$
- $imp(L) \cap \; exp(L) = \emptyset$

type checking (1)

general form of a linkset

- $L \equiv E_0 \mid x_1 \# \; E_1 \vdash a_1 : A_1 , \ldots, x_n \# \; E_n \vdash a_n : A_n$

linkset is intra-checked iff

- E_0 is well-formed
- each fragment is well typed in isolation

  $E_0 , E_i \vdash a_i : A_i$

type checking (2)

general form of a linkset

- $L \equiv E_0 \mid x_1 \# \ E_1 \vdash \ a_1 : A_1 , \ldots, x_n \# \ E_n \vdash \ a_n : A_n$

linkset is inter-checked iff

- it's intra-checked
- if E_i has the form E', x: A, E
  and x is x_j
  then A is A_j

type matching

- here requires exact match
- could use subtyping instead

linking

linking steps

- let $L \equiv E_0 \mid x\#(\varnothing \vdash a : A), \ldots, y\#(x{:}A', E \vdash \mathcal{J})$
- then $L \hookrightarrow E_0 \mid x\#(\varnothing \vdash a : A), \ldots, y\#(E \vdash \mathcal{J}[x \leftarrow a])$
  is a linking step
- write $L \hookrightarrow^* L'$ for reflexive transitive closure

note

- linking requires an empty environment
  so build bottom-up, and no cycles

algorithm

- keep doing linking steps until
  either linked (all environments except E_0 empty)
  or stuck (no further steps possible and some E_i not empty)

properties of linking

- termination
- compatibility
  if L and L’ are compatible linksets
  and L $\hookrightarrow^*$ L”
  then L’ and L” are compatible
- reduction soundness & completeness
  essentially that algorithm implements $\hookrightarrow^*$ maximally
- linking soundness
  if inter-checked(L) and L $\hookrightarrow$ L’
  then inter-checked(L’)

modules

key idea

- a simple fragment exports a value
- a module exports a binding
  client doesn't name the module itself!

bindings & signatures

- a binding is a list of definitions
  x : int = 3, …
- a signature is a list of declarations
  x : int

compilation

- a binding is like a linkset
- compilation transforms a binding into a linkset

separate compilation

key theorem

- given two modules whose interfaces are compatible
- their compiled linksets are inter-checked

key ideas

two phases

- intra-checking: module is well typed
- inter-checking: module interfaces match

linking criteria

- easy to understand and predict behaviour
- allow partial linking, order independent
- linking preserves well-formedness properties

theory assumes

- modules are outermost (no hiding)
- no mutual references
- import/export types match exactly