Program Fragments, Linking, and Modularization

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

POPL’97

Introduction

- Current module/class systems do not support well a basic requirement of software engineering: software development that is separate in time and space.
- How could we determine whether such a requirement is satisfied? We need a framework in which we can discuss the properties of the process that turns separate program fragments into whole programs. That process is *linking*.
- We aim to study:
  - ~ Separate typechecking and compilation of program fragments, including modules/classes.
  - ~ Type-correct linking of program fragments.

State of Affairs

- Anomalies in module systems.
  - ~ Module systems that do not support separate compilation (SML, some versions).
  - ~ Class systems where inherited methods must be retypechecked.
- Anomalies in development cycles.
  - ~ Separate compilation pitfalls exist at *every* step of the software development cycle; see paper introduction.

Type Safety

- Type safety for whole programs:

  A program that typechecks can be compiled in such a way that the resulting executable will not exhibit certain run-time errors.
- Type safety *for modular programs*:

  Program fragments that typecheck *and are compatible* can be compiled *and linked* in such a way that the resulting executable will not exhibit certain run-time errors.
- Linking is whatever process is needed to combine separately compiled fragments into bigger compiled fragments (libraries) or executables.

Inferences about Linking

- We would like to enable the formal description of inferences such as:
  - ~ If module M typechecks, then its compiled fragments (one or more) can be safely linked.
  - ~ If modules M_1 and M_2 separately typecheck and have compatible interfaces, then their compiled fragments can be merged and safely linked.
  - ~ If modules M_1, M_2, and M_3 separately typecheck and have compatible interfaces, then the compiled fragments of M_1, and M_2 can be safely pre-linked, and the result can be safely linked with the compiled fragments of M_3.
  - ~ Etc.

Program Fragments

- A *term judgment* represents a *program fragment*.

$$E \vdash a : A$$

The *environment* E contains type information about other fragments.

The *term* a is the program fragment in question.

The *type* A is the type of the fragment.

- In programming notation:

fragment
import E
export : A
begin a **end**.

- Examples:

$$\emptyset, x{:}Nat \vdash x{+}1 : Nat$$

$$\emptyset, f{:}Nat{\to}Nat \vdash \lambda(x{:}Nat)\, f(x){+}1 : Nat{\to}Nat$$

- N.B.:

The intended interpretation of $E \vdash a : A$ is that a represents a compiled code fragment, and E and A capture a's typing.

For simplicity, however, we let the object language coincide with the source language: a is a source term.

Even so, there will be a notion of compilation: the translation of *modules* to *linksets*.

Linksets

- A *linkset* is a collection of linkable fragments.
- It is represented by a *labeled collection of judgments*.

$$x_1 \mapsto E_1 \vdash a_1 : A_1$$
$$\ldots$$
$$x_n \mapsto E_n \vdash a_n : A_n$$

The x_i are names of fragments; they match the names in the E_j.

That is, the x_i (exports) and the E_j (imports) describe how the various fragments of a linkset plug together.

- N.B.:

Each linkset also has an environment E_0 that collects the global imports of the linkset. We skip this detail for now.

- Example:

$$f \mapsto (\emptyset \vdash \lambda(x{:}Nat)x : Nat{\rightarrow}Nat),$$
$$main \mapsto (\emptyset, f{:}Nat{\rightarrow}Nat \vdash f(3) : Nat)$$

- In programming notation:

```
fragment                   fragment
import nothing             import f : Nat→Nat
export f : Nat→Nat         export main : Nat
begin                      begin
    λ(x:Nat)x                  f(3)
end.                       end.
```

Linking

- *Substitution* represents *linking*.

  To perform a single linking step, we find two distinct labeled judgments in *L* of the form:

$$x \mapsto \emptyset \vdash a : A$$
$$y \mapsto \emptyset, x{:}A, E \vdash \mathfrak{I}$$

  and we replace the second labeled judgment as follows:

$$y \mapsto \emptyset, E \vdash \mathfrak{I}\{x{\leftarrow}a\}$$

  (The rest of the linkset remains the same.)

- A *linking algorithm* is a way of applying linking steps until no longer possible.

- Example:

$$f \mapsto (\emptyset \vdash \lambda(x{:}Nat)x : Nat{\rightarrow}Nat),$$
$$main \mapsto (\emptyset, f{:}Nat{\rightarrow}Nat \vdash f(3) : Nat)$$
$$\leadsto$$
$$f \mapsto (\emptyset \vdash \lambda(x{:}Nat)x : Nat{\rightarrow}Nat),$$
$$main \mapsto (\emptyset \vdash (\lambda(x{:}Nat)x)(3) : Nat)$$

No further linking: all environments are now empty.

- This view of linking is not totally accurate because:
  - ~ It expands code instead of threading it.

    But we could use *explicit substitutions* (a technique that represents substitutions symbolically and can delay expansion indefinitely).
  - ~ It works at the source level.

    But we can easily imagine the same mechanisms operating at the object code level. (In fact, λ-calculus is sometimes object code.)

    In any case, a linkset should be seen as the target of a translation. The source of the translation is a collection of modules.

Modules

- A *binding judgment* represents a *module*.

$$E \vdash d \therefore S$$

The *environment E* describes needed imports.

The *binding d* is a collection of definitions.

The *signature S* is the interface of the module.

- In programming notation:

module
import E
export S
begin d **end**.

- Example:

module
import nothing
export $x{:}Nat$
begin
$x : Nat = 3$
end.

module
import $x{:}Nat$
export $f{:}Nat{\rightarrow}Nat,\ m{:}Nat$
begin
$f : Nat{\rightarrow}Nat = \lambda(y{:}Nat)y{+}x,$
$m : Nat = f(x)$
end.

Those two modules are written as the two judgments:

$$\emptyset \ \vdash \ x{:}Nat{=}3, \emptyset \ \therefore \ x{:}Nat, \emptyset$$

$$\begin{gathered}\emptyset, x{:}Nat \vdash \\ f{:}Nat{\rightarrow}Nat{=}\lambda(y{:}Nat)y{+}x,\ m{:}Nat{=}f(x), \emptyset \\ \therefore\ f{:}Nat{\rightarrow}Nat,\ m{:}Nat, \emptyset\end{gathered}$$

The *import lists* are *environments*,

the *export lists* are *signatures*,

the *module bodies* are *bindings*.

Typing

Typing rules for F_1

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x{:}A \vdash \diamond}$$

(Type Const)
$$\frac{E \vdash \diamond}{E \vdash K}$$

(Type Arrow)
$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A{\rightarrow}B}$$

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

(Val Fun)
$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A{\rightarrow}B}$$

(Val Appl)
$$\frac{E \vdash b : A{\rightarrow}B \qquad E \vdash a : A}{E \vdash b(a) : B}$$

Signatures and Bindings for F_1

(Signature $\emptyset$)

$$\frac{E \vdash \diamond}{E \vdash \emptyset}$$

(Signature x)

$$\frac{E, x{:}A \vdash S}{E \vdash x{:}A, S}$$

(Binding $\emptyset$)

$$\frac{E \vdash \diamond}{E \vdash \emptyset \therefore \emptyset}$$

(Binding x)

$$\frac{E, x{:}A \vdash d \therefore S \qquad E \vdash a{:}A}{E \vdash (x{:}A{=}a,\ d) \therefore (x{:}A,\ S)}$$

Separate compilation

- Bindings can be (separately) compiled to linksets.

For example, the binding judgment:

$$\emptyset,\ x{:}Nat \vdash$$
$$f{:}Nat{\to}Nat{=}\lambda(y{:}Nat)y{+}x,\ m{:}Nat{=}f(x),\ \emptyset$$
$$\therefore\ f{:}Nat{\to}Nat,\ m{:}Nat,\ \emptyset$$

can be translated to the linkset

$$\emptyset,\ x{:}Nat\ \mid$$
$$f \mapsto \emptyset \vdash \lambda(y{:}Nat)y{+}x : Nat{\to}Nat,$$
$$m \mapsto \emptyset,\ f{:}Nat{\to}Nat \vdash f(x) : Nat$$

where the environment of the binding judgment ($\emptyset$, $x{:}Nat$) becomes a prefix for each environment in the linkset.

- The general form of the translation of bindings to linksets, $\langle\!\langle - \rangle\!\rangle$, is given by the following definition.

$$\langle\!\langle E \vdash d \therefore S \rangle\!\rangle \quad \triangleq \quad E \mid \langle\!\langle \emptyset \vdash d \therefore S \rangle\!\rangle^{\circ}$$

$$\langle\!\langle E \vdash \emptyset \therefore \emptyset \rangle\!\rangle^{\circ} \quad \triangleq \quad \textit{empty fragment list}$$
$$\langle\!\langle E \vdash (x{:}A{=}a,\ d) \therefore (x{:}A,\ S) \rangle\!\rangle^{\circ} \quad \triangleq$$
$$x \mapsto E \vdash a{:}A,\ \langle\!\langle E, x{:}A \vdash d \therefore S \rangle\!\rangle^{\circ}$$

Well-formedness conditions for linksets

- In general, a linkset L has the shape:

$$E_0 \ \mid\ x_1 \mapsto E_1 \vdash a_1 : A_1 \ \ \ldots\ \ x_n \mapsto E_n \vdash a_n : A_n$$

~ *linkset*(L) if (there are no trivial name clashes and):

each E_i is covered by the x_j

E_0 is disjoint from the x_j

~ *intra-checked*(L) if in addition:

$$E_0, E_i \vdash a_i : A_i \qquad \text{for each } i{\in}1..n$$

~ *inter-checked*(L) if in addition:

$$x_j{:}A \in E_i \ \Rightarrow\ A \equiv A_j \qquad \text{for each } i{\in}1..n$$

Properties

- Separate compilation produces good linksets:

  If $E \vdash d \therefore S$
  then *inter-checked*($\langle\!\langle E \vdash d \therefore S \rangle\!\rangle$).

- Linking preserves good linksets:

  If *inter-checked*(L) and $L \rightsquigarrow\!\!\twoheadrightarrow L'$
  then *inter-checked*(L').

(This property does not hold for *intra-checked*.)

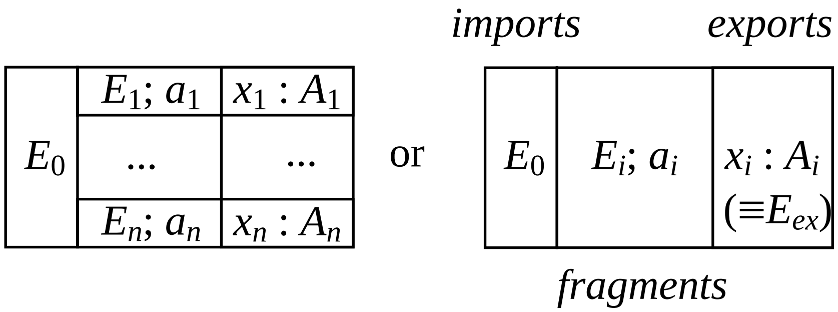
Linkset Merge

- Each modules is compiled to a linksets.
- In order to combine multiple modules into linkable entities, the corresponding linksets must be *merged*.

- Let's display a linkset

$$E_0 \;\mid\; x_1 \mapsto E_1 \vdash a_1 : A_1 \;\; \ldots \;\; x_n \mapsto E_n \vdash a_n : A_n$$

as:

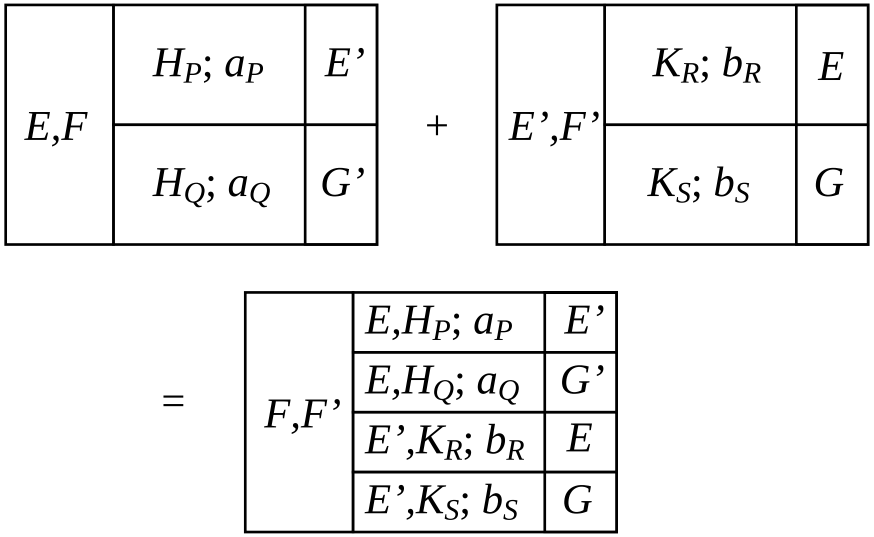


E_0	$E_1; a_1$	$x_1 : A_1$
	...	...
	$E_n; a_n$	$x_n : A_n$

or

	imports	*exports*
E_0	$E_i; a_i$	$x_i : A_i$ $(\equiv E_{ex})$
	fragments	

- Then the merge of two linksets is then defined as:



E,F	$H_P; a_P$	E'
	$H_Q; a_Q$	G'

+

E',F'	$K_R; b_R$	E
	$K_S; b_S$	G

=

F,F'	E,$H_P; a_P$	E'
	E,$H_Q; a_Q$	G'
	E',$K_R; b_R$	E
	E',$K_S; b_S$	G

Properties

- The linksets of separately compiled modules can be safely merged (and then safely linked):

  Assume $E \vdash d \therefore S$, $E' \vdash d' \therefore S'$,
  and $(E \vdash S) \div (E' \vdash S')$.
  Then, *inter-checked*$(\langle\!\langle E \vdash d \therefore S \rangle\!\rangle + \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle)$.

  Where $(E \vdash S) \div (E' \vdash S')$ iff $E \div E'$, $E \div S'$, $E' \div S$, and the domains of S and S' are disjoint.

  Where $E \div E'$ iff $E(x) = E'(x)$ for every x in the domain of both. Similarly for $E \div S$.

- Also in the paper:
  - Confluence of linking reductions.
  - A linking algorithm and its properties (termination, soundness, completeness).
  - A high-level inference system for separate compilation and linking.

Conclusions

- Reasoning about linking is becoming important. We have shown that linking can be reasonably formalized.
- Separate compilation can now be understood as the ability to translate separate modules to separate linksets (which are then merged and linked).
- Future directions:
  - More realistic formalization of linking.
  - More advanced module systems.
  - What about dynamic linking?