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.
  ```

**Linksets**

- A *linkset* is a collection of linkable fragments.

- It is represented by a *labeled collection of judgments*.
  
  $$ x_1 \equiv E_1 \vdash a_1 : A_1 
  
  \ldots 
  
  x_n \equiv 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.

**Examples:**

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

- 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.
Example:

\[
f \vdash (\varnothing \vdash \lambda(x: \text{Nat})x : \text{Nat} \rightarrow \text{Nat}),
\]

\[
\text{main} \vdash (\varnothing, f: \text{Nat} \rightarrow \text{Nat} \vdash f(3) : \text{Nat})
\]

In programming notation:

\[
\begin{align*}
\text{fragment} & \quad \text{fragment} \\
\text{import nothing} & \quad \text{import } f : \text{Nat} \rightarrow \text{Nat} \\
\text{export } f : \text{Nat} \rightarrow \text{Nat} & \quad \text{export } \text{main} : \text{Nat} \\
\text{begin} & \quad \text{begin} \\
\lambda(x: \text{Nat})x & \quad f(3) \\
\text{end.} & \quad \text{end.}
\end{align*}
\]

Linking

- Substitution represents linking.

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

\[
\begin{align*}
x & \vdash \varnothing \vdash a : A \\
y & \vdash \varnothing, x: A, E \vdash \emptyset
\end{align*}
\]

and we replace the second labeled judgment as follows:

\[
y \vdash \varnothing, E \vdash \emptyset\{x \leftarrow a\}
\]

(The rest of the linkset remains the same.)

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

Example:

\[
\begin{align*}
f & \vdash (\varnothing \vdash \lambda(x: \text{Nat})x : \text{Nat} \rightarrow \text{Nat}), \\
\text{main} & \vdash (\varnothing, f: \text{Nat} \rightarrow \text{Nat} \vdash f(3) : \text{Nat})
\end{align*}
\]

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, \( \lambda \)-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.
  
  $E \vdash d : S$

  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:

  \[
  \text{module}
  \text{import } E
  \text{export } S
  \begin{align*}
  &\text{begin } d \text{ end.}
  \end{align*}
  \]

Those two modules are written as the two judgments:

\[
\begin{align*}
\emptyset & \vdash x : \text{Nat} = 3, \emptyset : x : \text{Nat}, \emptyset \\
\emptyset, x : \text{Nat} & \vdash f : \text{Nat} \to \text{Nat} = \lambda(y : \text{Nat})y + x, m : \text{Nat} = f(x), \emptyset \\
& : f : \text{Nat} \to \text{Nat}, m : \text{Nat}, \emptyset
\end{align*}
\]

The import lists are environments,
the export lists are signatures,
the module bodies are bindings.

Typing

**Typing rules for $F_1$**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash A$</td>
<td>$x \notin \text{dom}(E)$</td>
</tr>
<tr>
<td>$E \vdash B$</td>
<td>$E \vdash A \to B$</td>
</tr>
<tr>
<td>$E \vdash K$</td>
<td>$E \vdash A \to B$</td>
</tr>
<tr>
<td>$E \vdash x : E(x)$</td>
<td>$E \vdash \lambda(x : A)b : A \to B$</td>
</tr>
<tr>
<td>$E \vdash b(a) : B$</td>
<td>$E \vdash b : A \to B$</td>
</tr>
</tbody>
</table>

Example:

\[
\begin{align*}
\text{module} & \quad \text{import } x : \text{Nat} \\
\text{import} & \quad \text{nothing} \\
\text{export} & \quad f : \text{Nat} \to \text{Nat}, m : \text{Nat} \\
\begin{align*}
\text{begin} & \quad x : \text{Nat} = 3 \\
& \quad f : \text{Nat} \to \text{Nat} = \lambda(y : \text{Nat})y + x, \\
& \quad m : \text{Nat} = f(x) \\
\text{end.}
\end{align*}
\]

\[
\begin{align*}
\text{module} & \quad \text{import } x : \text{Nat} \\
\text{export} & \quad f : \text{Nat} \to \text{Nat}, m : \text{Nat} \\
\begin{align*}
\text{begin} & \quad x : \text{Nat} = 3 \\
& \quad f : \text{Nat} \to \text{Nat} = \lambda(y : \text{Nat})y + x, \\
& \quad m : \text{Nat} = f(x) \\
\text{end.}
\end{align*}
\]
Signatures and Bindings for $F_1$

<table>
<thead>
<tr>
<th>Signature $\emptyset$</th>
<th>Signature $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash \emptyset$</td>
<td>$E, x : A \vdash S$</td>
</tr>
<tr>
<td>$E \vdash \emptyset$</td>
<td>$E \vdash x : A, S$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binding $\emptyset$</th>
<th>Binding $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash \emptyset$</td>
<td>$E, x : A \vdash d : S$ $E \vdash a : A$</td>
</tr>
<tr>
<td>$E \vdash \emptyset : \emptyset$</td>
<td>$E \vdash (x : A = a, d) : (x : A, S)$</td>
</tr>
</tbody>
</table>

Separate compilation

- Bindings can be (separately) compiled to linksets.

For example, the binding judgment:

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

$$\vdash : f : \text{Nat} \rightarrow \text{Nat}, \ m : \text{Nat}, \ \emptyset$$

can be translated to the linkset

$$\emptyset, x : \text{Nat} \mid$$

$$f \vdash \emptyset \vdash \lambda (y : \text{Nat}) y + x : \text{Nat} \rightarrow \text{Nat},$$

$$m \vdash \emptyset, f : \text{Nat} \rightarrow \text{Nat} \vdash f(x) : \text{Nat}$$

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

Well-formedness conditions for linksets

- In general, a linkset $L$ has the shape:

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

$\sim linkset(L)$ if (there are no trivial name clashes and):

- each $E_i$ is covered by the $x_j$
- $E_0$ is disjoint from the $x_j$

$\sim intra-checked(L)$ if in addition:

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

$\sim inter-checked(L)$ if in addition:

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

- Separate compilation produces good linksets:
  If \( E \vdash d : S \)
  then inter-checked(\( E \vdash d : S \))
- Linking preserves good linksets:
  If inter-checked(\( L \)) and \( L \rightsquigarrow L' \)
  then inter-checked(\( L' \)).
  (This property does not hold for intra-checked.)

Linkset Merge

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

Let's display a linkset

\[
E_0 \vdash x_1 \vdash E_1 \vdash a_1 : A_1 \ldots x_n \vdash E_n \vdash a_n : A_n
\]

as:

\[
\begin{array}{|c|c|c|}
\hline
\text{imports} & \text{exports} \\
\hline
E_0; a_1 & x_1 : A_1 \\
\vdots & \vdots \\
E_n; a_n & x_n : A_n \\
\hline
\end{array}
\]

\text{fragments}

Then the merge of two linksets is then defined as:

\[
\begin{array}{|c|c|c|}
\hline
E,F & H_P; a_P & E' \\
\hline
H_Q; a_Q & G' \\
\hline
\end{array} +
\begin{array}{|c|c|c|}
\hline
E',F' & K_R; b_R & E \\
\hline
K_S; b_S & G \\
\hline
\end{array} =
\begin{array}{|c|c|c|}
\hline
E,H_P; a_P & E' \\
\hline
E,H_Q; a_Q & G' \\
\hline
E',K_R; b_R & E \\
\hline
E',K_S; b_S & G \\
\hline
\end{array}
\]
Properties

- The linksets of separately compiled modules can be safely merged (and then safely linked):
  Assume $E \vdash d : S$, $E' \vdash d' : S'$,
  and $(E \vdash S) \div (E' \vdash S')$.
  Then, $\text{inter-checked}(\langle E \vdash d : S \rangle + \langle E' \vdash d' : S' \rangle)$.

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 $E \div S$.

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?

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.