Program Fragments, Linking, and Modularization

Luca Cardelli

luca@pa.dec.com

Digital Equipment Corporation, Systems Research Center

Abstract

Module mechanisms have received considerable theoretical attention, but the associated concepts of separate compilation and linking have not been emphasized. Anomalous module systems have emerged in functional and object-oriented programming where software components are not separately typecheckable and compilable. In this paper we provide a context where linking can be studied, and separate compilability can be formally stated and checked. We propose a framework where each module is separately compiled to a self-contained entity called a *linkset*; we show that separately compiled, compatible modules can be safely linked together.

1 Introduction

Program modularization arose from the necessity of splitting large programs into fragments in order to compile them. As system libraries grew in size, it became essential to compile the libraries separately from the user programs; libraries acquired interfaces that minimized compilation dependencies. A linker was used to patch compiled fragments together.

It was soon realized that modularization had great advantages in terms of large-grain program structuring [19]. Much fundamental and practical research focused on modularization principles and properties; milestones in this process are embodied in such constructs as object-oriented classes, Modula-2 modules, and Standard ML functors.

Since program structuring is of great importance in software engineering, there is motivation for continuously increasing the flexibility and convenience of modularization constructs. Unfortunately, in the shadow of many exciting developments there has been a tendency to overlook the original purpose of modularization. Some language definitions specify what are to be the compilation units (e.g.: Ada [12]), but others do not (e.g.: Standard ML [17]). A paradoxical question then arises: when does a module system *really* support modularization (meant as separate compilation)?

Appears in the Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

In designing and formalizing module systems, many proposals have focused on the analogy between modules and data structures, and between interfaces and data types, e.g. as in Burstall's influential paper [4]. In such proposals, modules and interfaces become language constructs to program with. This approach has the advantage of adding clean programmability to the area of system configuration, where it has traditionally been lacking. When pushing this approach to extremes, though, there is the danger of losing sight of the requirements of separate compilation.

In this paper we take a different approach in order to maintain a natural and accurate view of the separate compilation and linking process. We consider linking as the fundamental process from which module mechanisms arise: not merely as a technique for managing large programs and libraries. Further, we consider modularization as inseparable from separate compilation: not merely as a program structuring mechanism. Instead of considering interfaces as just another program construct, we look at interfaces as typing environments that are intrinsically external to the programming language. By adopting this view we can develop modularization mechanisms with precise notions of separate compilation, inter-module typechecking, and linking.

Today, the purpose of separate compilation is to be able to write, check, deliver, maintain, and upgrade libraries of code, possibly hiding the source code from the clients of the libraries. Many things can go wrong in languages and environments designed (or coerced) to support separate compilation. To understand the range of problems that may arise, let us consider an example of a software development cycle and the obstacles that may impede it. In this example, a library module and a user module interact over time; it is instructive to assume that library development and client development happen in separate locations.

Day 1: Library description. A library interface I_{Lib} is publicized before any corresponding implementation module M_{Lib} is made available. The purpose is to allow early development of client software that will later be integrated with the library. Therefore, in this scenario we assume that there exists a notion of largely code-free interfaces.

Obstacles • Early programming languages, both procedural and object-oriented, did not separate interfaces from implementations. • Languages that are designed to be "small" or untyped often lack interfaces. • Certain language features may require global analysis and may thus conflict with modularity; examples are multimethods [7] and overloading.

Day 2: User program description. A user interface I_{Usr} is written without yet producing the corresponding user module M_{Usr}. The purpose is to begin designing the structure of M_{Usr} and its interaction with I_{Lib} before making any actual implementation commitments. The interface I_{Usr} is based on I_{Lib}.

Obstacles • It is important to be able to write I_{Usr} on the basis of definitions contained in I_{Lib}; the purpose of I_{Lib} is often to define shared types. Modula-2, for example, allows type definitions in interfaces. However, this feature has surprisingly complex interactions with the type theory of modules, and even advanced module systems like Standard ML's did not consider it until recently [11, 13].

Day 3: User program compilation. A user module M_{Usr} is written and compiled. It is checked to be compatible with I_{Usr} and I_{Lib}. The compilation of M_{Usr} produces a linkable image L_{Usr}. No running program is generated yet because no implementation of I_{Lib} has been delivered.

Obstacles • The code of generic library modules may have to be instantiated before the user code can be typechecked; then an implementation M_{Lib} of I_{Lib} must be available to typecheck M_{Usr}. • The instantiation of generic interfaces and modules performed by the client may produce unexpected type errors in the library code that were not detected by simple testing of the library [18, page 47]. • Some object-oriented languages need to retypecheck superclass code (potentially library code) to verify the correct use of Self-types in subclasses [20]. • Standard ML's transparent signatures [15] allow situations where M_{Usr} depends on the types defined in a particular implementation of I_{Lib}; therefore M_{Usr} cannot be isolated by I_{Lib} from that implementation [13]. • Even when it is possible to typecheck M_{Usr} purely against I_{Lib}, it may be that I_{Lib} does not convey all the information necessary to produce a linkable image L_{Usr}. For example: the compiler may insist on performing global flow analysis, or some routines of I_{Lib} may require inlining, or the layout of opaque types in I_{Lib} may have to be determined.

Day 4: Library compilation. A library module M_{Lib} is produced that matches the interface I_{Lib}. It is compiled to a linkable image L_{Lib}. The pair (I_{Lib}, L_{Lib}) is stored in a public repository.

Obstacles • It may be the case that a library cannot be compiled even though its full code is available. This happens for generic module mechanism in the style on templates (as in C++, ADA, and Modula-3) where generic library modules must be instantiated by client (or test) modules before typechecking can take place.

Day 5: User program linking. The user fetches the archived library L_{Lib} associated with I_{Lib} from the repository. A user program P_{Usr} is produced by linking L_{Lib} with L_{Usr}.

Obstacles • Even though M_{Lib} matches I_{Lib}, M_{Usr} matches I_{Usr}, and I_{Usr} matches I_{Lib}, it may be the case that P_{Usr} produces run-time type errors. In Eiffel, for example, separate typechecking of classes does not imply that the whole program is type-safe [8, 16]. • Some mechanisms (Modula-3's revelations [18], Standard ML's smartest recompilation [21], Eiffel's proposed link-time safety analysis) delay some type checks until link time: the user may discover at that point internal inconsistencies in the libraries. • The linked program should have the same effect as a program obtained by merging all the sources together and compiling the result in a single step. Such a merging of sources is not often characterized; then the semantics of linking is undetermined.

Day 6: Library implementation evolution. A new library module M'_{Lib} that matches I_{Lib} is produced. A new pair (I_{Lib}, L'_{Lib}) is stored in the public repository.

Obstacles • Changes to the implementation of a library superclass may alter object layout; this may require recompilation of user subclasses, even when the public interface of the superclass does not change. • When many interdependent libraries are archived, there may be transients when the library implementations in the repository are mutually inconsistent, and when the linking of user programs should fail.

Day 7: User program relinking. The user program P_{Usr} is now out of date, but I_{Lib} has not changed. Therefore, a new user program P'_{Usr} can be regenerated without recompilation by linking L'_{Lib} with L_{Usr}.

Obstacles • Will the result of running the relinked program be the same as if it had been recompiled first? It is natural to expect so. However, David Griswold [10] has pointed out that this property fails for Java (without compromising type safety), because overloading is treated differently during compilation and linking.

Day 8: Library interface evolution. A revised interface $\tilde{I}_{Lib}$ and a corresponding library module $\tilde{M}_{Lib}$ are generated. A new pair ($\tilde{I}_{Lib}$, $\tilde{L}_{Lib}$) is stored in the public repository, replacing (I_{Lib}, L'_{Lib}).

Obstacles • When many such interdependent libraries are archived, there may be transients when the library interfaces in the repository are mutually inconsistent and when the compilation of user code should fail.

Day 9: User program adaptation. Because of the new $\tilde{M}_{Lib}$, the user program is now out of date. Moreover, M_{Usr} and I_{Usr} do not match $\tilde{I}_{Lib}$. Thus, I_{Usr} is changed to a compatible $\tilde{I}_{Usr}$, and a new $\tilde{M}_{Usr}$ is compiled to $\tilde{L}_{Usr}$. Finally, a new $\tilde{P}_{Usr}$ is produced by linking $\tilde{L}_{Lib}$ with $\tilde{L}_{Usr}$.

Obstacles • Unless code dependencies are tracked properly [1], the new version of the user code, $\tilde{L}_{Usr}$, may be accidentally linked with the old library, L_{Lib}, (or vice versa) causing arbitrary execution errors even in safe languages.

As discussed in this scenario, the potential and actual problems in separate compilation and linking are many and varied. Moreover, the example sketched above concerns mostly traditional environments. Linkers are now getting smarter, taking advantage of type information at link time and performing dynamic linking at run time. As an emerging issue, security in Java depends not only on safe typing, but also on safe linking [9]. Thus the potential for problems is increasing.

We do not propose to attack all the obstacles at once: some have to do with language design, some with implementation technology, and some with environment engineering. However, it should be clear that separate compilation and linking have become complex enough that they require very careful thinking, and possibly formal thinking. At every point in the software development process we would like to be confident that our programs are correctly linked. This work is meant as a formal step in this direction, mostly concerning the interactions of linking with type safety.

In this paper we make a number of simplifying assumptions in the attempt to render the technical development as rigorous and simple as possible. Our main intent is to provide a road map for more ambitious efforts concerning realistic module systems. For concreteness and simplicity, we apply our ideas to a simple module system for a first-order language ($\mathbf{F_1}$) and we hint at possible extensions.

Section 2 introduces basic terminology about separate compilation and proposes a formal interpretation of linking. Sections 3 and 4 review the simply typed λ-calculus and introduce a simple module system for it. Sections 5 and 6 study linksets and linking algorithms. Section 7 maps modules to linksets and section 8 establishes a reasoning system for the soundness of separate compilation and linking. Finally, section 9 draws some conclusions and discusses future work and extensions.

2 Linking

In programming environments, linking is the process that turns a collection of program fragments into a runnable program. In this section we discuss the formalization of linking in terms of the manipulation of judgments.

2.1 Program Fragments

A *program fragment* is, in first approximation, any syntactically well-formed program term, possibly containing free variables. *Separate compilation* is intended as the separate typechecking and separate code generation of program fragments. We avoid issues of code generation by always working at the source-language level, even when discussing linking. Therefore, compilation is simplified to typechecking. (We believe this is not an important restriction for our purposes: the hardest part of separate compilation is separate typechecking, at least from the point of view of language design.)

A program fragment cannot be compiled (or typechecked) in isolation, but it can be compiled in the context of adequate information about missing fragments. This information is usually given in terms of an environment for the free variables of a fragment. The notion of a *typing environment* E for a program fragment a is routinely employed in the formalization of typability; a judgment $E \vdash a{:}A$ establishes a type A for the program fragment a with respect to the environment E.

The *separate compilation* of a fragment a can be seen as the compilation of a judgment $E \vdash a{:}A$, because the judgment contains sufficient (although incomplete) information about related fragments. During the compilation of this judgment, the types of the free variables of a are found in E (without any associated values). Since the values of free variables are missing, the compilation is incomplete, but can still be carried out *separately*, i.e., modulo the missing values.

A *complete program* is a closed term; that is, a term with no free variables. A complete program is self-contained: it can be typed in an empty environment, and its compilation can be carried out completely.

In programming environments, the linking process is used to produce a complete program from a collection of program fragments. In addition, linking is used to combine a number of program fragments without necessarily forming a complete program. The result of such an incomplete linking is called a *library*: in its original meaning, it is a library of routines to be used by other programs. Libraries can be linked again to form larger libraries or complete programs. A consistent (in ways to be determined) collection of linkable program fragments is called a *software system*, or simply a *system*.

Separate compilation, in our framework, maps judgments $E \vdash a{:}A$ into entities we call *linksets*, over which we can define linking operations. We can see the judgments $E \vdash a{:}A$ as the source module language, and the linksets as the target language of the compilation. The module language is in this case very rudimentary, but our approach extends to other module languages. In Section 4 we consider a more complex module language, and in Section 7 we compile it to linksets.

2.2 A Simple Configuration Language

The linking process starts from a collection of program fragments, and from a description of how the fragments should be combined. This description is traditionally expressed in a *configuration language*, whose complexity can range from simple file-naming conventions to sophisticated scripts. These scripts have been named *project files*, *makefiles*, *system models*, etc.

We are going to investigate the simple configuration language of *linksets*, where a collection of fragments to be linked is expressed as a collection of named judgments:

$E_0 \mid x_1 \mapsto E_1 \vdash \Im_1 \ldots x_n \mapsto E_n \vdash \Im_n$

This is a *linkset*, consisting of an environment E_0 and a collection of judgments $E_i \vdash \Im_i$, each named by a label x_i. The components $x_i \mapsto E_i \vdash \Im_i$ are called linkset fragments.

The main intuitions are that (1) E_0 is the external interface of the entire linkset (E_0 being non-empty for a library, and being empty for a complete program), (2) the environment of each judgment is implicitly prefixed by E_0, so that $E_0, E_i \vdash \Im_i$ is a valid judgment, and (3) each judgment is labeled by a unique name x_i; these names match the free variables of other fragments, and thus determine how the fragments hook up[1]. Well-formedness conditions for linksets are discussed in detail later.

A conventional name, such as *main*, can be reserved for a judgment that denotes a complete program. The following is a linkset consisting of a single fragment called *main*. (In our initial examples we take E_0=$\emptyset$.)

$$\emptyset \mid main \mapsto (\emptyset \vdash 3{+}1 : Nat)$$

More interestingly, here is a linkset consisting of two fragments:

$$\begin{array}{l} \emptyset \mid \\ f \mapsto (\emptyset \vdash \lambda(x{:}Nat)x : Nat{\to}Nat), \\ main \mapsto (\emptyset, f{:}Nat{\to}Nat \vdash f(3) : Nat) \end{array}$$

In verbose programming notation, this linkset might be rearranged and written:

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

In this notation, the fragment *f* has an empty import list, and produces a value *f* of type *Nat→Nat*. The fragment *main* imports a fragment named *f* producing a value of type *Nat→Nat*, and produces a value of type *Nat*.

As we said, the linking strategy for linksets is specified by choosing names for fragments that correspond to the free variables of other fragments. Above, it is intended that the fragment named *f* provides a value for the free variable *f* of the fragment *main*. We can say that "*main needs f* " (the environment for *main* contains the assumption *f*:*Nat→Nat*) while "*f needs* nothing" (the environment for *f* is empty). This implicit *needs* relation partially specifies a dependency, or linking order, for the judgments in the linkset.

There are two main activities we can perform on linksets: checking the name and type information in a linkset, and performing the actual linking process. We consider these in turn.

The checking activity guarantees that the names and the types are used consistently within and across judgments, so that typing can be ignored in the subsequent linking phase; this corresponds to *intra-module* typechecking and to *inter-module* typechecking. In the *f,main* example above, the *intra-fragment* typechecking consists in checking, for example, that the term λ(*x*:*Nat*)*x* has the type exported by the fragment *f*. The *inter-fragment* typechecking consists in checking that the type of the *f* fragment matches the type of the *f* import of the *main* fragment.

The linking activity corresponds, technically, to the repeated application of substitutions. It assumes that all the typing requirements have been satisfied in the previous checking phase. In the example, we can eliminate the *f* assumption in the *main* judgment by substituting *f* with λ(*x*:*Nat*)*x*, and obtaining:

$$\begin{array}{l} \emptyset \mid \\ f \mapsto \emptyset \vdash \lambda(x{:}Nat)x : Nat{\to}Nat, \\ main \mapsto \emptyset \vdash (\lambda(x{:}Nat)x)(3) : Nat \end{array}$$

Since the typing environments of all the fragments are now empty, no other substitutions are possible. We have completed the linking process for this example; the relevant outcome is the fully linked *main* program.

The linking process may fail in some situations, in the sense of not being able to empty all environments. For example, the following linkset does not provide a fragment for *y*, so the *x* fragment cannot be fully linked:

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

We will rule out such incomplete linksets.

A more subtle case of linking failure is due to cyclic dependencies among fragments. The following linkset is not obviously incomplete, but it still cannot generate a runnable program because of a cyclic dependency of its single fragment with itself:

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

Problems with cycles become worse with fragments that are mutually dependent, as in the following linkset:

$$\begin{array}{l} \emptyset \mid \\ x \mapsto \emptyset, y{:}Nat \vdash y{-}1 : Nat, \\ y \mapsto \emptyset, x{:}Nat \vdash x{+}1 : Nat \end{array}$$

Conceivably, we could eliminate the cycles by converting them into fixpoints. The earlier linkset could be reduced to:

$$\emptyset \mid x \mapsto \emptyset \vdash \mu(x{:}Nat)x{+}1 : Nat$$

However, we prefer not to go down this road in this paper. The circumstances under which cyclic dependencies are acceptable depend strongly on specific languages, and are hard to generalize. Moreover, in this paper we will be handling in depth only a simply typed λ-calculus that is strongly normalizing; hence fixpoints would be out of character. We simply let the linking process fail (but not diverge) when presented with cycles. In other terms, we rule out recursive and mutually recursive modules.

3 The Simply Typed λ-calculus, $\mathbf{F_1}$

We now begin formalizing the intuitions of the previous section. We start with a description of system $\mathbf{F_1}$, a standard simply typed λ-calculus. In the following sections we define linksets for $\mathbf{F_1}$.

The types and terms of $\mathbf{F_1}$ have the following syntax. The types are either a base type *K* or function types. The terms are either variables, abstractions, or applications.

Syntax of $\mathbf{F_1}$

$A,B ::= K \mid A{\to}B$	types
$a,b ::= x \mid \lambda(x{:}A)b \mid b(a)$	terms

[1] Alternatively, one could distinguish between program variables that can be freely α-converted and associated labels that connect the fragments, as in [11].

We use a single uninterpreted base type K, but we could easily enrich $\mathbf{F_1}$ with base types such as *Bool* and *Nat*.

The environments E of $\mathbf{F_1}$ are lists of typing assumptions of the form $\emptyset, x_1{:}A_1, ..., x_n{:}A_n$ for $n\geq 0$; the empty environment is $\emptyset$. We use the notations $dom(E)$, $(E, x{:}A)$, (E, E'), $env(E)$, and $E(x)$:

Definition 3-1 (Environment operations)

- $dom(\emptyset, x_1{:}A_1, ..., x_n{:}A_n) \triangleq \{x_i{}^{i\in 1..n}\}$.
- If $E \equiv \emptyset, x_1{:}A_1, ..., x_n{:}A_n$ and $E' \equiv \emptyset, y_1{:}B_1, ..., y_m{:}B_m$, then $E, x_{n+1}{:}A_{n+1} \triangleq \emptyset, x_1{:}A_1, ..., x_n{:}A_n, x_{n+1}{:}A_{n+1}$ and $E, E' \triangleq \emptyset, x_1{:}A_1, ..., x_n{:}A_n, y_1{:}B_1, ..., y_m{:}B_m$.
- $env(\emptyset, x_1{:}A_1, ..., x_n{:}A_n) \Leftrightarrow$ for all $i,j\in 1..n$, $i\neq j \Rightarrow x_i\neq x_j$.
- If E has the shape $E', x{:}A, E''$ and $env(E)$, then $E(x) \triangleq A$.

□

The type rules of $\mathbf{F_1}$ are given below. They are based on three judgments: $E \vdash \diamond$ (E is well-formed), $E \vdash A$ (type A is well-formed in E), and $E \vdash a : A$ (term a has type A in E).

Typing rules for $\mathbf{F_1}$

(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)
$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A\to B}$$

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

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

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

We list some standard technical lemmas. Here $\Im$ is any judgment right-hand-side (including $\diamond$), and $\Im\{x\leftarrow a\}$ is the substitution of a for the free occurrences of x in $\Im$. The notions of free and λ-bound occurrences are the standard ones. A technical note: we identify terms up to consistent renaming of bound variables, but we do not identify judgments up to renaming of environment variables. A judgment-renaming lemma can be proved, but will not be necessary here.

Lemma 3-2 ($\mathbf{F_1}$ properties)

- ***Implied Judgments.*** If $E, E' \vdash \Im$ then $E \vdash \diamond$. If $E, x{:}A, E' \vdash \Im$ then $E \vdash A$. If $E \vdash a : A$ then $E \vdash A$.
- ***Weakening***. If $E, E' \vdash \Im$ and $E, F \vdash \diamond$ and $dom(F) \cap dom(E') = \emptyset$ then $E, F, E' \vdash \Im$.
- ***Exchange***. If $E, F, F', E' \vdash \Im$ then $E, F', F, E' \vdash \Im$.
- ***Substitution***. If $E, x{:}A, E' \vdash \Im$ and $E \vdash a : A$ then $E, E' \vdash \Im\{x\leftarrow a\}$.

□

From these lemmas we easily obtain the following *linking lemma*, which states the essential conditions under which a linking step can be performed. Here, the program fragment a with environment E_1, E_2 is linked into the "hole" x of $\Im$, adapting the environment $E_1, x{:}A, E_3$ of $\Im$ to E_1, E_2, E_3.

Lemma 3-3 (Linking)

If $E_1, x{:}A, E_3 \vdash \Im$ and $E_1, E_2 \vdash a : A$
and $dom(\emptyset, x{:}A, E_3) \cap dom(E_2) = \emptyset$,
then $E_1, E_2, E_3 \vdash \Im\{x\leftarrow a\}$.

Proof

Assume $E_1, x{:}A, E_3 \vdash \Im$ and $E_1, E_2 \vdash a : A$ with $dom(\emptyset, x{:}A, E_3) \cap dom(E_2) = \emptyset$. By Implied Judgments we have $E_1, E_2 \vdash \diamond$, and by Weakening we obtain $E_1, E_2, x{:}A, E_3 \vdash \Im$. Finally, by Substitution we obtain $E_1, E_2, E_3 \vdash \Im\{x\leftarrow a\}$.

□

4 Simple Modules for $\mathbf{F_1}$

As described in the introduction, a judgment $E \vdash a : A$ can be seen as a simple module. In this section we explore a slightly more structured module system for $\mathbf{F_1}$, which corresponds to the following programming notation:

```
module                  module
import nothing          import x:Nat
export x:Nat            export f:Nat→Nat, m:Nat
begin                   begin
    x : Nat = 3,            f : Nat→Nat = λ(y:Nat)y+x
end.                        m : Nat = f(x)
                        end.
```

This is one of the simplest conceivable module systems for a programming language. A module has a list of imports and a list of exports. The body of a module contains definitions for its exports. Note that there is no mechanism for naming collections of imports or exports: lists of variables and their types are used explicitly. This module mechanism is only a small step forward from the program fragments of section 2.2, but at least it supports the grouping of related definitions. A similar mechanism was used in early versions of Modula.

We extend $\mathbf{F_1}$ with two new judgments for modularization; the way this extension is carried out is quite uniform, and can be applied to many type systems [4, 5, 14]. From our basic judgments we produce a *signature judgment* that represents export lists, and a *binding judgment* that represents modules. A *signature* is essentially a tuple of declarations, and is similar to an environment. A *binding* is essentially a tuple of definitions.

The signature judgment is written $E \vdash S$ (i.e., signature S is well-formed in E); the binding judgment is written $E \vdash d \mathrel{\therefore} S$ (i.e., binding d has signature S in E).

Signatures and Bindings for $\mathbf{F_1}$

(Signature $\emptyset$)
$$\frac{E \vdash \diamond}{E \vdash \emptyset}$$

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

(Binding ϕ) (Binding x)

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

According to these rules, in $E \vdash d \therefore S$ every component of d is matched by a component of S in the same position. One could allow signatures and bindings to match more flexibly, up to reasonable permutations and elisions of components, by instrumenting the rules above. Note that signatures and environments associate differently; nonetheless, we sometimes identify a signature $(x_1{:}A_1, ..., (x_n{:}A_n, \phi)..)$ with an environment $(..(\phi, x_1{:}A_1), ..., x_n{:}A_n)$.

The two program modules shown at the beginning of this section can be represented by the two binding judgments below. The import lists become environments, the export lists become signatures, and the module bodies become bindings.

$$\phi \vdash (x{:}Nat{=}3, \phi) \therefore (x{:}Nat, \phi)$$
$$\phi, x{:}Nat \vdash (\, f{:}Nat{\to}Nat{=}\lambda(y{:}Nat)y{+}x,\ m{:}Nat{=}f(x), \phi)$$
$$\therefore (\, f{:}Nat{\to}Nat,\ m{:}Nat, \phi)$$

5 Linksets

As we discussed in the introduction, a linkset is a collection of named judgments plus an interface. We now define linksets formally, and we describe a number of conditions that identify well behaved linksets. We begin with some terminology.

Definition 5-1 (Linkset structure)

Consider the structure $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i{}^{i\in 1..n}$, where each $\Im_i$ has the shape $a_i : A_i$.

Let *imp*$(L) \triangleq$ *dom*(E_0) be the *imported names* of L.

Let *exp*$(L) \triangleq \{x_1, ..., x_n\}$ be the *exported names* of L.

Let *names*$(L) \triangleq$ *imp*$(L) \cup$ *exp*(L) be the *names* of L.

Let *imports*$(L) \triangleq E_0$ be the *import environment* of L.

Let *exports*$(L) \triangleq \phi, x_1{:}A_1, ..., x_n{:}A_n$ be the *export environment* of L.

□

We first need to identify linksets that use names coherently, without worrying yet about any of the type information. The predicate *linkset*(L), defined below, captures this kind of coherence, which is the minimum required to perform linking. Recall that the predicate *env*(E), from Definition 3-1, asserts that the variables of E are distinct.

Definition 5-2 (Linksets)

Consider the structure $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i{}^{i\in 1..n}$.

linkset$(L) \Leftrightarrow$

- *env*(*imports*(L)), and *env*(*exports*(L))
- for all $i\in 1..n$, we have *env*(E_0, E_i) and *dom*$(E_i) \subseteq$ *exp*(L)
- *imp*$(L) \cap$ *exp*$(L) = \phi$.

□

Note that by the condition *dom*$(E_i) \subseteq$ *exp*(L), L is complete, in the sense that every assumption $x{:}A$ in one of the environments E_i is matched by a fragment named x. (Any missing fragment must be declared in E_0.) This completeness condition, however, does not guarantee the absence of cyclic dependencies.

We say that a linkset L is *linked* if all the E_i are empty, and is *fully linked* if, in addition, E_0 is empty.

We now define a predicate that refines *linkset* by performing additional checking. This corresponds to the amount of checking performed by separate compilation, before inter-module checking. The following definition of the predicate *intra-checked* guarantees that each judgment in a linkset is valid in $\mathbf{F_1}$, and that all the judgments have the common prefix E_0. The *intra-checked* predicate does not guarantee that the fragments hook up properly with each other with respect to typing.

Definition 5-3 (Intra-checked linksets)

Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i{}^{i\in 1..n}$.

intra-checked$(L) \Leftrightarrow$

- *linkset*(L)
- $E_0 \vdash \diamond$ and, for all $i\in 1..n$, we have $E_0, E_i \vdash \Im_i$.

□

We now turn to checking the consistency of linkset fragments with respect to each other. These checks, corresponding to inter-module typechecking, guarantee that the fragments forming the linkset can be linked in a type-safe way.

Definition 5-4 (Inter-checked linksets)

Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i{}^{i\in 1..n}$.

inter-checked$(L) \Leftrightarrow$

- *intra-checked*(L)
- for all $j,k\in 1..n$, x, A, E', E'',
  if E_k has the form $E', x{:}A, E''$ and $x \equiv x_j$ then $A \equiv A_j$.

□

Here we require exact agreement between the fragments ($A \equiv A_j$). This definition may need to be refined in systems more complex than $\mathbf{F_1}$, for example for subtyping.

Each linkset includes an environment E_0 that is meant to describe the fragments that are missing from the linkset. Therefore, a useful operation on linksets is to combine two of them to mutually reduce the number of missing fragments. This operation produces a new linkset that is the *merge* of the two. We first need some operations on environments:

Definition 5-5 (Environment compatibility and merge)

- $E \backslash X$ is the environment obtained from E by removing the assumptions $x{:}A$ such that $x\in X$.
- $E \restriction X$ is environment obtained from E by retaining only the assumptions $x{:}A$ such that $x\in X$.
- Compatible environments: $E_1 \div E_2 \Leftrightarrow$ for all $x\in$ *dom*$(E_1) \cap$ *dom*(E_2) we have $E_1(x) = E_2(x)$.

- We define the merge of two environments E_1 and E_2 as $E_1{+}E_2 \triangleq E_1, (E_2 \backslash dom(E_1))$.

□

Lemma 5-6 (Commutation of environment merge)
If $E_1 \div E_2$ and $E, (E_1{+}E_2), E' \vdash \Im$ then $E, (E_2{+}E_1), E' \vdash \Im$.

Proof

From Lemma 3-2 (exchange), since $E_1{+}E_2$ is just a permutation of $E_2{+}E_1$ under the assumption $E_1 \div E_2$.

□

The merge of two linksets is then defined as follows. The imports of the two linksets are merged, except that the fragments mutually exported are removed from the combined imports. Then, the exported fragments are merged; the environment of each fragment of a linkset is enriched with the imports of that linkset that are exported by the other linkset.

Definition 5-7 (Linkset merge)
Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i\ {}^{i \in 1..n}$, $L' \equiv E_0' \mid x_i' \mapsto E_i' \vdash \Im_i'\ {}^{i \in 1..n'}$.
If $linkset(L)$, $linkset(L')$, and $exp(L) \cap exp(L') = \emptyset$, then:
$L{+}L' \triangleq$
$E_0 \backslash exp(L') + E_0' \backslash exp(L) \mid$
$x_i \mapsto E_0 \upharpoonright exp(L'), E_i \vdash \Im_i\ {}^{i \in 1..n}$,
$x_i' \mapsto E_0' \upharpoonright exp(L), E_i' \vdash \Im_i'\ {}^{i \in 1..n'}$

□

The following lemmas show that the merge of two linksets preserves the properties *linkset*, *intra-checked*, and *inter-checked*, under appropriate assumptions. The proofs are given in Appendix.

Lemma 5-8 (Linkset merge)
If $linkset(L)$, $linkset(L')$, and $exp(L) \cap exp(L') = \emptyset$,
then $linkset(L{+}L')$.

□

Lemma 5-9 (Intra-checked merge)
If $intra\text{-}checked(L)$, $intra\text{-}checked(L')$,
$imports(L) \div imports(L')$, and $exp(L) \cap exp(L') = \emptyset$,
then $intra\text{-}checked(L{+}L')$.

□

Definition 5-10 (Linkset compatibility)
$L \div L' \iff$
$imports(L) \div imports(L')$, $imports(L) \div exports(L')$,
$imports(L') \div exports(L)$, and $exp(L) \cap exp(L') = \emptyset$.

□

Lemma 5-11 (Inter-checked merge)
If $inter\text{-}checked(L)$, $inter\text{-}checked(L')$, and $L \div L'$,
then $inter\text{-}checked(L{+}L')$.

□

6 Linking

A linkset L contains a set of interdependent fragments of the form $x_i \mapsto E_i \vdash \Im_i\ {}^{i \in 1..n}$. The purpose of linking is to resolve the dependencies by making all the E_i empty via substitutions.

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

$x \mapsto \emptyset \vdash a{:}A$
$y \mapsto x{:}A', E \vdash \Im$

and we replace the second labeled judgment as follows (without requiring $A \equiv A'$):

$y \mapsto E \vdash \Im\{x \leftarrow a\}$

Formally, a linking step $L \rightsquigarrow L'$ transforms a linkset L into a linkset L' by performing a single substitution:

Definition 6-1 (Linking steps)
Let $L \equiv (E_0 \mid ..., (x \mapsto \emptyset \vdash a{:}A), ..., (y \mapsto x{:}A', E \vdash \Im), ...)$ and assume $linkset(L)$.

- $L \rightsquigarrow (E_0 \mid ..., (x \mapsto \emptyset \vdash a{:}A), ..., (y \mapsto E \vdash \Im\{x \leftarrow a\}), ...)$ is a linking step.
- We write $L \not\rightsquigarrow$ if there is no L' such that $L \rightsquigarrow L'$.
- We write $\rightsquigarrow\!\!\!\!\rightsquigarrow$ for the reflexive and transitive closure of $\rightsquigarrow$.

□

This definition of linking step imposes a rather strict order of reductions by requiring one of the environments involved to be empty. One could relax this restriction, and allow more flexible linking orders (such generalizations are supported by Lemma 3-3). However, we adopt the simpler definition.

Linking steps preserve the *linkset* and *inter-checked* properties:

Lemma 6-2 (Properties preserved by linking steps)
(1) If $linkset(L)$ and $L \rightsquigarrow L'$ then $linkset(L')$.
(2) If $inter\text{-}checked(L)$ and $L \rightsquigarrow L'$ then $inter\text{-}checked(L')$.

Proof

(1) Easy, from the definition of *linkset*, since the *env* property is preserved by shortening environments, and $names(L) = names(L')$.

(2) Consider $L \equiv E_0 \mid x_i \mapsto E_i \vdash a_i{:}A_i\ {}^{i \in 1..n}$. Suppose the reduction is carried out on the pair $(x \mapsto \emptyset \vdash a{:}A)$, $(y \mapsto x{:}A', E \vdash \Im)$. Since $inter\text{-}checked(L)$ by assumption, we have $A \equiv A'$, and since $intra\text{-}checked(L)$, we have $E_0 \vdash a{:}A$ and $E_0, x{:}A, E \vdash \Im$. By Lemma 3-3 (linking), we have $E_0, E \vdash \Im\{x \leftarrow a\}$. Therefore, we have $intra\text{-}checked(L')$, since $E_0, E \vdash \Im\{x \leftarrow a\}$ is the only new fragment in L'. Moreover, we have *inter-checked*(L'), since the environments in L' are the same as the ones in L except for one that becomes shorter, and the

A_i are the same (the substitution $\Im\{x\leftarrow a\}$ does not affect types).

□

However, *intra-checked*(L) and $L \rightsquigarrow L'$ do not imply *intra-checked*(L'). As should be expected, intra-checking of fragments is not sufficient for performing linking soundly.

We state two important properties of *linking reductions* (sequences of linking steps). (1) The *inter-checked* property is preserved by reductions, meaning that linking does not violate typing. (2) Reductions are confluent, meaning that linking steps can be performed in any order.

Proposition 6-3 (Subject reduction for linking)

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

Proof

By Lemma 6-2, the *inter-checked* property is preserved at every step.

□

Proposition 6-4 (Linking is confluent)

Assume *linkset*(L). If $L \rightsquigarrow\!\!\!\!\rightarrow L_1$ and $L \rightsquigarrow\!\!\!\!\rightarrow L_2$ then there exists an L_3 such that $L_1 \rightsquigarrow\!\!\!\!\rightarrow L_3$ and $L_2 \rightsquigarrow\!\!\!\!\rightarrow L_3$.

Proof

(1) We first show that if $L \rightsquigarrow L_1$ and $L \rightsquigarrow L_2$, then either $L_1 = L_2$ or there exists an L_3 such that $L_1 \rightsquigarrow L_3$ and $L_2 \rightsquigarrow L_3$. Moreover, *linkset*(L_1), *linkset*(L_2), and *linkset*(L_3).

By the assumption *linkset*(L) we know that fragment names in L are distinct.

Consider two linking steps $L \rightsquigarrow L_1$ and $L \rightsquigarrow L_2$ of the form:

$(E_0 \mid ..., (x_1 \mapsto \emptyset \vdash a_1{:}A_1), ..., (y_1 \mapsto x_1{:}A_1', E_1 \vdash \Im_1), ...)$

$\rightsquigarrow (E_0 \mid ..., (x_1 \mapsto \emptyset \vdash a_1{:}A_1), ..., (y_1 \mapsto E_1 \vdash \Im_1\{x_1\leftarrow a_1\}), ...)$

$(E_0 \mid ..., (x_2 \mapsto \emptyset \vdash a_2{:}A_2), ..., (y_2 \mapsto x_2{:}A_2', E_2 \vdash \Im_2), ...)$

$\rightsquigarrow (E_0 \mid ..., (x_2 \mapsto \emptyset \vdash a_2{:}A_2), ..., (y_2 \mapsto E_2 \vdash \Im_2\{x_2\leftarrow a_2\}), ...)$

By Lemma 6-2 we have *linkset*(L_1) and *linkset*(L_2). Let us consider all possible identifications of x_1, y_1, x_2, and y_2.

We have $x_1 \neq y_1$, $x_2 \neq y_2$, $x_1 \neq y_2$, and $x_2 \neq y_1$, because of the shape of the associated environments.

If $y_1 = y_2$ we also have $x_1 = x_2$, by the shape of the associated environments. Then we trivially have $L_1 = L_2$.

If $y_1 \neq y_2$ (and either $x_1 = x_2$ or $x_1 \neq x_2$), the two linking steps do not interfere since they affect distinct fragments, and we can trivially find an L_3 (containing the y_1 fragment from L_1 and the y_2 fragment from L_2) such that $L_1 \rightsquigarrow L_3$ and $L_2 \rightsquigarrow L_3$. By Lemma 6-2 we have *linkset*(L_3).

(2) From (1) we can easily show that the reflexive closure $\rightsquigarrow^R$ of $\rightsquigarrow$ is *confluent*: if $L \rightsquigarrow^R L_1$ and $L \rightsquigarrow^R L_2$ then there exists an L_3 such that $L_1 \rightsquigarrow^R L_3$ and $L_2 \rightsquigarrow^R L_3$. (Moreover, *linkset*($L_i$) for $i \in \{1,2,3\}$). The transitive closure of a confluent relation is also confluent, by a standard "tiling" argument. Therefore, $\rightsquigarrow\!\!\!\!\rightarrow$ is confluent.

□

We can now define a simple *linking algorithm* that applies linking steps until no longer possible. The algorithm attempts to bring the linkset into the *linked* state, where the environments of all the fragments are empty.

Algorithm 6-5 (Link)

Assuming *linkset*(L), the algorithm *Link* with input L produces an output $\langle L', r\rangle$ (if it terminates) where $r \in \{$*success*, *failure*$\}$. The algorithm iterates from the initial L:

```
If $L \not\rightsquigarrow$, then
    if linked($L$) then exit with $\langle L, success\rangle$,
    else exit with $\langle L, failure\rangle$.
Else, choose any linking step $L \rightsquigarrow L'$,
set $L := L'$, and repeat.
```

□

We can show that the linking algorithm terminates, and that it is sound and complete with respect to linking reductions. We also obtain that linking can be performed soundly on *inter-checked* linksets.

Proposition 6-6 (Link properties)

- ***Termination.*** If *linkset*(L), then the algorithm *Link* terminates over the input L.
- ***Compatibility.*** If *linkset*(L), *linkset*(L'), $L \div L'$, and *Link*(L) terminates with $\langle L'', r\rangle$, then $L'' \div L'$.
- ***Reduction Soundness.*** If *linkset*(L) and *Link*(L) terminates with $\langle L', r\rangle$, then $L \rightsquigarrow\!\!\!\!\rightarrow L'$.
- ***Reduction Completeness.*** If *linkset*(L), $L \rightsquigarrow\!\!\!\!\rightarrow L'$, and $L' \not\rightsquigarrow$, then *Link*(L) $= \langle L', r\rangle$ with $r \in \{$*success*, *failure*$\}$.
- ***Linking Soundness.*** If *inter-checked*(L) then *Link*(L) $= \langle L', r\rangle$ for some L' and r, and *inter-checked*(L').

Proof

Termination. The algorithm performs a finite number of iterations, because at every iteration either the length of an environment is reduced or the algorithm exits.

Compatibility. Linking steps preserve the sets *imports*(L) and *exports*(L). Therefore compatibility is preserved by *Link*.

Reduction Soundness. This follows by definition of *Link*. At each step of the iteration the *linkset* property is preserved by Lemma 6-2.

Reduction Completeness. *Link*(L) terminates; assume it produces $\langle L'', r\rangle$. By soundness, $L \rightsquigarrow\!\!\!\!\rightarrow L''$. By confluence, there exists an L_0 such that $L' \rightsquigarrow\!\!\!\!\rightarrow L_0$ and $L'' \rightsquigarrow\!\!\!\!\rightarrow L_0$. However, $L' \not\rightsquigarrow$, by assumption, hence we must have $L' \equiv L_0$. Moreover, $L'' \not\rightsquigarrow$, by the exit condition of the algorithm, hence we must have $L'' \equiv L_0$. Therefore, $L'' \equiv L'$.

Linking Soundness. By Termination and Reduction Soundness we obtain $L \rightsquigarrow\!\!\!\rightarrow L'$. Then, by Proposition 6-3 we obtain *inter-checked*(L').

□

7 Modules as Linksets

In this section we consider the module system for $\mathbf{F_1}$ of section 4, and we prove safe-linking properties for it. A binding, like a linkset, is a collection of fragments. Therefore, it is natural to compile bindings 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 following linkset, where the environment of the binding judgment ($\emptyset$, $x{:}Nat$) becomes the interface of the linkset:

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

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

Definition 7-1 (Compilation of a binding)

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

□

The following lemma details the correspondence between a binding judgment and its corresponding linkset. (Note: we confuse signatures with environments.)

Lemma 7-2 (Properties of compilation)
If $L \equiv \langle\!\langle E \vdash d \therefore S \rangle\!\rangle$ then $imports(L) = E$ and $exports(L) = S$.

Proof

Clearly, $imports(\langle\!\langle E \vdash d \therefore S \rangle\!\rangle) = imports(E \mid \langle\!\langle \emptyset \vdash d \therefore S \rangle\!\rangle^{\circ}) = E$.

If $L \equiv E \mid U$, let $exports(U) \triangleq exports(L)$. We prove that if $U \equiv \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle^{\circ}$ then $exports(U) = S'$, by induction on the translation $\langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle^{\circ}$.

Case $\langle\!\langle E' \vdash \emptyset \therefore \emptyset \rangle\!\rangle^{\circ}$. We have $U \equiv \langle\!\langle E' \vdash \emptyset \therefore \emptyset \rangle\!\rangle^{\circ} \equiv$ *empty fragment list*. Hence, $exports(U) = exports(E \mid U) = \emptyset = S'$.

Case $\langle\!\langle E' \vdash (x{:}A{=}a, d'') \therefore (x{:}A, S'') \rangle\!\rangle^{\circ}$. We have $U \equiv x \mapsto E' \vdash a{:}A, U'$ and $U' \equiv \langle\!\langle E', x{:}A \vdash d'' \therefore S'' \rangle\!\rangle^{\circ}$. By induction hypothesis we have $exports(U') = S''$. Hence, $exports(U) = x{:}A, S''$.

□

We can now state the first important property of separate compilation: well-typed modules are compiled to well-typed linksets.

Theorem 7-3 (Separate compilation)
If $E \vdash d \therefore S$ then *inter-checked*($\langle\!\langle E \vdash d \therefore S \rangle\!\rangle$).

Proof

The translation $\langle\!\langle E \vdash d \therefore S \rangle\!\rangle$ produces a structure of the shape $L \equiv E \mid x_i \mapsto E_i \vdash \Im_i{}^{i \in 1..n}$. We have $dom(S) = dom(exports(L)) = exp(L)$, and $dom(E) = dom(imports(L)) = imp(L)$.

(1) We first show *linkset*(L) and *intra-checked*(L) by checking the necessary conditions. We have that $imports(L) = E$, and since $E \vdash d \therefore S$, we have $E \vdash \diamond$ and $env(E)$. Since $exports(L) = S$ and $E \vdash d \therefore S$, we have $env(S)$. By induction on the derivation of $E \vdash d \therefore S$, for all $i \in 1..n$, we have $E, E_i \vdash \Im_i$, and $env(E, E_i)$. By construction, each E_i is a prefix of S, hence $dom(E_i) \subseteq dom(S) = exp(L)$. By construction, $dom(E) \cap dom(S) = \emptyset$; that is, $imp(L) \cap exp(L) = \emptyset$.

(2) To show that *inter-checked*(L), we note that, by the definition of $\langle\!\langle - \rangle\!\rangle^{\circ}$, every time an assertion $x{:}A$ is added to an environment, a fragment of the form $x \mapsto E' \vdash a{:}A$ is added to the linkset.

□

The second important property of separate compilation is that two well-typed modules with compatible interfaces can be safely compiled and merged. For this, we first need to define the notion of compatibility of signature and binding judgments:

Definition 7-4 (Signature and binding compatibility)

$$(E \vdash S) \div (E' \vdash S') \;\triangleq \\ E \div E', E \div S', E' \div S, \text{ and } dom(S) \cap dom(S') = \emptyset.$$
$$(E \vdash S) \div (E' \vdash d' \therefore S') \;\triangleq\; (E \vdash S) \div (E' \vdash S')$$
$$(E \vdash d \therefore S) \div (E' \vdash S') \;\triangleq\; (E \vdash S) \div (E' \vdash S')$$
$$(E \vdash d \therefore S) \div (E' \vdash d' \therefore S') \;\triangleq\; (E \vdash S) \div (E' \vdash S')$$

□

Lemma 7-5 (Compatibility under compilation)
Assume $(E \vdash d \therefore S) \div (E' \vdash d' \therefore S')$.
Then, $\langle\!\langle E \vdash d \therefore S \rangle\!\rangle \div \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle$.

Proof

By definition 7-4, $(E \vdash d \therefore S) \div (E' \vdash d' \therefore S')$ implies $(E \vdash S) \div (E' \vdash S')$, and hence $E \div E'$, $E \div S'$, $E' \div S$, and $dom(S) \cap dom(S') = \emptyset$. Take $L \equiv \langle\!\langle E \vdash d \therefore S \rangle\!\rangle$ and $L' \equiv \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle$. By Lemma 7-2, $imports(L) \div imports(L')$, $imports(L) \div exports(L')$, $exports(L) \div imports(L')$, and $exp(L) \cap exp(L') = \emptyset$. Therefore, by Definition 5-10, $L \div L'$.

□

We now show that compatibility of signatures is a sufficient condition for the safe merge of separately compiled modules:

Theorem 7-6 (Separate compilation and merge)
Assume $E \vdash d \therefore S$, $E' \vdash d' \therefore S'$, and $(E \vdash S) \div (E' \vdash S')$.
Then, *inter-checked*($\langle\!\langle E \vdash d \therefore S \rangle\!\rangle + \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle$).

Proof

Let $L \equiv \langle\!\langle E \vdash d \therefore S \rangle\!\rangle$ and $L' \equiv \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle$. By Theorem 7-3 we have *inter-checked*(L) and *inter-checked*(L'). Since $(E \vdash S) \div (E' \vdash S')$, we also have $(E \vdash d \therefore S) \div (E' \vdash d' \therefore S')$ by Definition 7-4. By Lemma 7-5 we obtain $L \div L'$. Therefore, by Lemma 5-11 we have *inter-checked*(L+L').

□

Note that the linking of $\langle\!\langle E \vdash d \therefore S \rangle\!\rangle + \langle\!\langle E' \vdash d' \therefore S' \rangle\!\rangle$ may still produce *failure*, because cyclic dependencies may be present between $E \vdash d \therefore S$ and $E' \vdash d' \therefore S'$.

8 Summary

We can summarize our main definitions and results by recasting them as an inference system for establishing the soundness of sequences of compilation and linking steps.

In the following inference rules, M is a module represented by a binding judgment $E \vdash d \therefore S$, and L is a linkset; $\langle\!\langle M \rangle\!\rangle$ is the compilation of a module to a linkset. By *valid*(M) we mean that M is derivable (type-consistent). $M \div M'$ is type-compatibility between modules (or their interfaces). By *inter-checked*(L) we mean that L is type-consistent. $L \div L'$ is type-compatibility and $L + L'$ is merging of linksets. By *link*(L) = L' we mean that the linking algorithm *Link*(L) yields $\langle L', r \rangle$ where r is *success* or *failure.* The relations indicated by $\div$ are symmetric.

Separate compilation inference system

(Compilation) (Theorem 7-3)

$$\frac{\mathit{valid}(M)}{\mathit{inter\text{-}checked}(\langle\!\langle M \rangle\!\rangle)}$$

(Compilation compatibility) (Lemma 7-5)

$$\frac{\mathit{valid}(M) \qquad \mathit{valid}(M') \qquad M \div M'}{\langle\!\langle M \rangle\!\rangle \div \langle\!\langle M' \rangle\!\rangle}$$

(Linking) (Proposition 6-6)

$$\frac{\mathit{inter\text{-}checked}(L)}{\mathit{inter\text{-}checked}(\mathit{link}(L))}$$

(Linking compatibility) (Proposition 6-6)

$$\frac{\mathit{inter\text{-}checked}(L) \qquad \mathit{inter\text{-}checked}(L') \qquad L \div L'}{\mathit{link}(L) \div L'}$$

(Merge) (Lemma 5-11)

$$\frac{\mathit{inter\text{-}checked}(L) \qquad \mathit{inter\text{-}checked}(L') \qquad L \div L'}{\mathit{inter\text{-}checked}(L+L')}$$

From these rules, we can show that a separately compiled valid module M can be safely partially linked. That is, that *inter-checked*(*link*($\langle\!\langle M \rangle\!\rangle$)) holds:

valid(M)	assumption	
$\Rightarrow$ *inter-checked*($\langle\!\langle M \rangle\!\rangle$)	by (Compilation)	
$\Rightarrow$ *inter-checked*(*link*($\langle\!\langle M \rangle\!\rangle$))	by (Linking)	(1)

Furthermore, the following derivation shows that two separately compiled compatible valid modules M and M', one of which has been partially linked, can be safely linked together. That is, that *inter-checked*(*link*(*link*($\langle\!\langle M \rangle\!\rangle$)+$\langle\!\langle M' \rangle\!\rangle$)) holds:

valid(M')	assumption	
$\Rightarrow$ *inter-checked*($\langle\!\langle M' \rangle\!\rangle$)	by (Compilation)	(2)
valid(M), *valid*(M'), $M \div M'$	assumptions	
$\Rightarrow \langle\!\langle M \rangle\!\rangle \div \langle\!\langle M' \rangle\!\rangle$	by (Compil. compat.)	(3)
link($\langle\!\langle M \rangle\!\rangle$) $\div \langle\!\langle M' \rangle\!\rangle$	by (1), (2), (3), (Linking compat.)	(4)
inter-checked(*link*($\langle\!\langle M \rangle\!\rangle$)+$\langle\!\langle M' \rangle\!\rangle$)	by (1), (2), (4), (Merge)	
$\Rightarrow$ *inter-checked*(*link*(*link*($\langle\!\langle M \rangle\!\rangle$)+$\langle\!\langle M' \rangle\!\rangle$))	by (Linking)	

Thus, inference systems such as the one outlined here can be used to check the validity of complex sequences of compilation and linking steps, at a reasonable level of abstraction.

9 Conclusions

The linking process, once obscure and undocumented, is becoming increasingly visible and sophisticated. In some instances, it is becoming part of language semantics.

We suggest that linking and separate compilation should be seriously taken into account when designing a language and module system. This sentence may seem a truism, but these issues have been surprisingly under-emphasized in the technical literature. We have shown that linking can be given a technical content. We have formalized linking via linksets, and we have formalized separate compilation as the ability to translate modules separately to linksets that can be safely linked. The general intuition is to regard linking as the repeated application of type-preserving substitutions.

We have studied a simplistic module system. It should be possible to use the same basic ideas to explore other module mechanisms, hopefully more realistic ones. Many directions of further work are possible, including the following: • Alternative linking reductions and algorithms. • Linking algorithms that handle mutual dependencies. • A more realistic linking process that does not cause code expansion or loss of module identity (by using explicit substitutions [2]). • Convenient naming of module interfaces, and support for the dot notation [6]. • Flexible signature matching and subtyping. • Linking and separate compilation for the polymorphic λ-calculus, $\mathbf{F_2}$, with the aim of covering the modularization features of Modula-2. • Design of advanced module systems that are nonetheless able to support separate compilation [3, 13]. • Study of dynamic linking.

Acknowledgments

The anonymous referees provided valuable suggestions and criticism.

Appendix

Lemma 5-8

If *linkset*(L), *linkset*(L'), and $exp(L) \cap exp(L') = \emptyset$, then *linkset*($L+L'$).

Proof

Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i \ ^{i\in 1..n}$ and $L' \equiv E_0' \mid x_i' \mapsto E_i' \vdash \Im_i' \ ^{i\in 1..n'}$. We verify the conditions require by *linkset*($L+L'$), from Definition 5-2.

(1) From $env(E_0)$ and $env(E_0')$ we have $env(E_0\backslash exp(L') + E_0'\backslash exp(L))$, by definition of +. That is, $env(imports(L+L'))$.

(2) Since the x_i are distinct, and the x_i' are distinct, and because of the assumption $exp(L) \cap exp(L') = \emptyset$, we have that all the x_i, x_i' are distinct. That is, $env(exports(L+L'))$.

(3) Since $dom(E_i') \subseteq exp(L')$ and $dom(E_i) \subseteq exp(L)$, we have that $dom(E_0{\restriction}exp(L'), E_i) = dom(E_0{\restriction}exp(L')) \cup dom(E_i) \subseteq exp(L') \cup exp(L) = exp(L+L')$. Similarly, $dom(E_0'{\restriction}exp(L), E_i') \subseteq exp(L+L')$.

(4) We have $imp(L+L') = dom(E_0\backslash exp(L') + E_0'\backslash exp(L)) = (imp(L)–exp(L')) \cup (imp(L')–exp(L))$, and $exp(L+L') = exp(L) \cup exp(L')$. By assumption, we have $imp(L) \cap exp(L) = \emptyset$ and $imp(L') \cap exp(L') = \emptyset$. Now, $(imp(L)–exp(L')) \cap exp(L) = \emptyset$ and $(imp(L)–exp(L')) \cap exp(L') = \emptyset$, therefore $(imp(L)–exp(L')) \cap (exp(L) \cup exp(L')) = \emptyset$. Similarly, $(imp(L')–exp(L)) \cap (exp(L) \cup exp(L')) = \emptyset$. Hence $((imp(L)–exp(L')) \cup (imp(L')–exp(L))) \cap (exp(L) \cup exp(L')) = \emptyset$. That is, $imp(L+L') \cap exp(L+L') = \emptyset$.

(5) We need to show that:

$$env(E_0\backslash exp(L') + E_0'\backslash exp(L), E_0{\restriction}exp(L'), E_i)$$
$$env(E_0\backslash exp(L') + E_0'\backslash exp(L), E_0'{\restriction}exp(L), E_i')$$

From the assumptions $env(E_0, E_i)$ and $env(E_0', E_i')$ we trivially have $env(E_0{\restriction}exp(L'), E_i)$ and $env(E_0'{\restriction}exp(L), E_i')$. Moreover, case (1) shows $env(E_0\backslash exp(L') + E_0'\backslash exp(L))$. So, we are left to show that $dom(E_0\backslash exp(L') + E_0'\backslash exp(L)) = imp(L+L')$ is disjoint from both $dom(E_0{\restriction}exp(L'), E_i)$ and $dom(E_0'{\restriction}exp(L), E_i')$. Now, case (3) shows that the latter two are included in $exp(L+L')$, and case (4) shows that $imp(L+L') \cap exp(L+L') = \emptyset$. Therefore, we are done.

□

Lemma 5-9

If *intra-checked*(L), *intra-checked*(L'),
$imports(L) \div imports(L')$, and $exp(L) \cap exp(L') = \emptyset$,
then *intra-checked*($L+L'$).

Proof

Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i \ ^{i\in 1..n}$ and $L' \equiv E_0' \mid x_i' \mapsto E_i' \vdash \Im_i' \ ^{i\in 1..n'}$.

(1) By Lemma 5-8, we have *linkset*($L+L'$).

(2) We need to show that:

$$E_0\backslash exp(L') + E_0'\backslash exp(L), E_0{\restriction}exp(L'), E_i \vdash \Im_i$$

By Lemma 3-2 (implied judgments) we have $E_0 \vdash \diamond$ and $E_0' \vdash \diamond$, from which:

$$E_0\backslash exp(L'), (E_0'\backslash exp(L))\backslash dom(E_0\backslash exp(L')) \vdash \diamond$$

As in Lemma 5-8(5), since $dom((E_0'\backslash exp(L))\backslash dom(E_0\backslash exp(L')) \subseteq dom(E_0\backslash exp(L') + E_0'\backslash exp(L))$:

$$dom((E_0'\backslash exp(L))\backslash dom(E_0\backslash exp(L')) \cap dom(E_0{\restriction}exp(L'), E_i) = \emptyset$$

By Lemma 3-2 (exchange), from $E_0, E_i \vdash \Im_i$:

$$E_0\backslash exp(L'), E_0{\restriction}exp(L'), E_i \vdash \Im_i$$

By Lemma 3-2 (weakening), from the previous three results:

$$E_0\backslash exp(L'), (E_0'\backslash exp(L))\backslash dom(E_0\backslash exp(L')), E_0{\restriction}exp(L'), E_i \vdash \Im_i$$

This is the same as:

$$E_0\backslash exp(L') + E_0'\backslash exp(L), E_0{\restriction}exp(L'), E_i \vdash \Im_i$$

(3) We need to show also that:

$$E_0\backslash exp(L') + E_0'\backslash exp(L), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$$

Or equivalently, by Lemma 5-6, since $imports(L) \div imports(L')$, that:

$$E_0'\backslash exp(L) + E_0\backslash exp(L'), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$$

By Lemma 3-2 (implied judgments) we have $E_0 \vdash \diamond$ and $E_0' \vdash \diamond$, from which:

$$E_0'\backslash exp(L), (E_0\backslash exp(L'))\backslash dom(E_0'\backslash exp(L)) \vdash \diamond$$

As in Lemma 5-8(5), since $dom((E_0\backslash exp(L'))\backslash dom(E_0'\backslash exp(L)) \subseteq dom(E_0\backslash exp(L') + E_0'\backslash exp(L))$:

$$dom((E_0\backslash exp(L'))\backslash dom(E_0'\backslash exp(L)) \cap dom(E_0'{\restriction}exp(L), E_i') = \emptyset$$

By Lemma 3-2 (exchange), from $E_0', E_i' \vdash \Im_i'$:

$$E_0'\backslash exp(L), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$$

By Lemma 3-2 (weakening), from the previous three results:

$$E_0'\backslash exp(L), (E_0\backslash exp(L'))\backslash dom(E_0'\backslash exp(L)), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$$

i.e. $E_0'\backslash exp(L) + E_0\backslash exp(L'), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$

By Lemma 5-6, since $imports(L) \div imports(L')$ we conclude:

$$E_0\backslash exp(L') + E_0'\backslash exp(L), E_0'{\restriction}exp(L), E_i' \vdash \Im_i'$$

□

Lemma 5-11

Assume *inter-checked*(L), *inter-checked*(L'), *imports*(L) $\div$ *imports*(L'), *imports*(L) $\div$ *exports*(L'), *imports*(L') $\div$ *exports*(L), and *exp*(L) $\cap$ *exp*(L') = $\emptyset$.
Then *inter-checked*(L+L').

Proof

Let $L \equiv E_0 \mid x_i \mapsto E_i \vdash \Im_i\ ^{i\in 1..n}$ and $L' \equiv E_0' \mid x_i' \mapsto E_i' \vdash \Im_i'\ ^{i\in 1..n'}$.

(1) By Lemma 5-9, we have *intra-checked*(L+L').

(2) We have the following fragments for L+L':

$x_i \mapsto E_0{\restriction}exp(L'), E_i \vdash \Im_i\ ^{i\in 1..n}$,
$x_i' \mapsto E_0'{\restriction}exp(L), E_i' \vdash \Im_i'\ ^{i\in 1..n'}$

By assumption, we know that:

1) If E_i has the form F, x:A, G then there exists a j (since $dom(E_i) \subseteq exp(L)$) with $x \equiv x_j$ and $A \equiv A_j$.

2) If E_i' has the form F', x':A', G' then there exists a j (since $dom(E_i') \subseteq exp(L')$) with $x' \equiv x_j'$ and $A' \equiv A_j'$.

We need to show that for any assumption z:C appearing in $E_0{\restriction}exp(L')$, E_i or $E_0'{\restriction}exp(L)$, E_i', if there is a fragment named z in L+L', it has type C.

For any assumption in E_i, and E_i' the hypotheses apply.

For an assumption x:A in $E_0{\restriction}exp(L')$, we have that E_0=*imports*(L) $\div$ *exports*(L'). Hence x:A is in *exports*(L'), which means that there is an $x_j' \equiv x$ with $A_j' \equiv A$.

For an assumption x':A' in $E_0'{\restriction}exp(L)$, we have that E_0' = *imports*(L') $\div$ *exports*(L). Hence x':A' is in *exports*(L), which means that there is an $x_j \equiv x'$ with $A_j \equiv A'$.

□

References

[1] Abadi, M., J.-J. Levy, and B. Lampson, **Analysis and caching of dependencies**. *Proc. 1996 ACM International Conference on Functional Programming*, 83-91. 1996.

[2] Abadi, M., L. Cardelli, P.-L. Curien, and J.-J. Lévy, **Explicit substitutions**. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*. 1990.

[3] Appel, A.W. and D.B. MacQueen, **Separate compilation for Standard ML**. *Proc. 1994 ACM Conf. on Programming Language Design and Implementation, (ACM SIGPLAN Notices vol. 29, number 6)*, 13-23, June 1994.

[4] Burstall, R.M., **Programming with modules as typed functional programming**. *Proc. International Conference on 5th Generation Computing Systems*. Tokyo. 1984.

[5] Cardelli, L., **Typeful programming**. In *Formal Description of Programming Concepts*, E.J. Neuhold and M. Paul, ed. Springer-Verlag. 431-507. 1991.

[6] Cardelli, L. and X. Leroy, **Abstract types and the dot notation**. *Proc. Programming Concepts and Methods*, 479-504. North Holland. 1990.

[7] Chambers, C. and G.T. Leavens, **Typechecking and modules for multi-methods**. *ACM Transactions on Programming Languages and Systems* **17**(6), 805-843. 1995.

[8] Cook, W.R., **A proposal for making Eiffel type-safe**. *Proc. European Conference of Object-Oriented Programming*, 57-72. 1989.

[9] Dean, D., Personal communication. July 1996.

[10] Griswold, D. *et al.*, **Fundamental flaw in Java library distribution scheme**. comp.lang.java thread, November 1995.

[11] Harper, R. and M. Lillibridge, **A type-theoretic approach to higher-order modules with sharing**. *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*. 123-137, 1994.

[12] Ichbiah, J., J.G.P. Barnes, J.C. Heliard, B. Krieg-Bruecker, O. Roubine, and B.A. Wichmann, **Rationale for the design of the ADA programming language**, *ACM SIGPLAN Notices* **14**(6), 1979.

[13] Leroy, X., **Manifest types, modules, and separate compilation**. *Proc. 21st ACM Symposium on Principles of Programming Languages*, 109-122. 1994.

[14] Leroy, X., **A modular module system**. *Research report 2866*, INRIA. April 1996.

[15] MacQueen, D.B., **Using dependent types to express modular structure**. *Proc. 13th Annual ACM Symposium on Principles of Programming Languages. 277-286.* 1986.

[16] Meyer, B., **Typing issues in object-oriented programming**. *Invited address, ACM Conference on Object Oriented Programming Systems, Languages, and Applications* 1995. Interactive Software Engineering Inc. 1995.

[17] Milner, R., M. Tofte, and R. Harper, **The definition of Standard ML**. MIT Press. 1989.

[18] Nelson, G., ed. **Systems programming with Modula-3**. Prentice Hall. 1991.

[19] Parnas, D.L., **On the criteria to be used in decomposing systems into modules**. *Communications of the ACM* **15**(12), 1053-1058. 1972.

[20] Schaffert, C., T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, **An introduction to Trellis/Owl**. *Proc. ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, 9-16. 1986.

[21] Shao, Z. and A.W. Appel, **Smartest recompilation**. *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, 439-450. 1993.