A Calculus for Link-time Compilation

Elena Machkasova[1] and Franklyn A. Turbak[2⋆]

[1] elenam@bu.edu, Boston University, Boston MA 02215, USA
[2] fturbak@wellesley.edu, Wellesley College, Wellesley MA 02481, USA

Abstract. We present a module calculus for studying a simple model of link-time compilation. The calculus is stratified into a term calculus, a core module calculus, and a linking calculus. At each level, we show that the calculus enjoys a *computational soundness* property: if two terms are equivalent in the calculus, then they have the same outcome in a small-step operational semantics. This implies that any module transformation justified by the calculus is meaning preserving. This result is interesting because recursive module bindings thwart confluence at two levels of our calculus, and prohibit application of the traditional technique for showing computational soundness, which requires confluence. We introduce a new technique, based on properties we call *lift* and *project*, that uses a weaker notion of *confluence with respect to evaluation* to establish computational soundness for our module calculus. We also introduce the *weak distributivity property* for a transformation T operating on modules D_1 and D_2 linked by $\oplus$: $T(D_1 \oplus D_2) = T(T(D_1) \oplus T(D_2))$. We argue that this property finds promising candidates for link-time optimizations.

1 Introduction

We present a module calculus for a purely functional language that is a tool for exploring the design space for a simple form of link-time compilation. Link-time compilation lies in the relatively unexplored expanse between *whole-program compilation*, in which the entire source program is compiled to an executable, and *separate compilation*, in which source program modules are independently compiled into fragments, which are later linked to form an executable. In the link-time compilation model (1) source program modules are first partially compiled into intermediate language modules; (2) intermediate modules are further compiled when they are combined, taking advantage of usage information exposed by the combination; and (3) when all intermediate modules have been combined into a final closed module, it is translated into an executable.

Link-time compilation can potentially provide more reusability than whole-program compilation and more efficiency than separate compilation. While separate compilation offers well-known benefits for program development and code reuse, a drawback is that the compilation of one module cannot take advantage of usage information in the modules with which it is later linked. In contrast,

⋆ Both authors were supported by NSF grant EIA–9806747. This work was conducted as part of the Church Project (http://www.cs.bu.edu/groups/church/).

link-time compilation can use this information to perform optimizations and choose specialized data representations more efficient than the usual uniform representations for data passed across module boundaries.

In this paper we take some first steps towards formalizing link-time compilation. There are three main contributions of this work. First, we present a stratified untyped call-by-value module calculus that at every level satisfies a *computational soundness* property[1]: if two expressions can be shown equivalent via calculus steps, then their outcomes relative to a small-step operational semantics will be observably equal. This implies that any transformation expressible as a sequence of calculus steps (such as constant propagation and folding, function inlining, and many others) is meaning preserving.

Second, our technique for proving soundness is interesting in its own right. Traditional techniques for showing this property (e.g., [Plo75,AF97]) require the language to be confluent, but the recursive nature of module bindings destroys confluence. In order to show that our module calculus has soundness, we introduce a new technique for proving this property based on a weaker notion we call *confluence with respect to evaluation*. We replace the confluence and standardization of the traditional technique for proving soundness with symmetric properties we call *lift* and *project*.

Third, we sketch a simple model of link-time compilation and introduce the *weak distributivity* property as one way to find candidates for link-time optimizations. We show that module transformations satisfying certain conditions are weakly distributive, and demonstrate these conditions for some examples of meaning preserving transformations.

Our work follows a long tradition of using untyped calculi for reasoning about programming languages features: e.g., call-by-name vs. call-by-value semantics [Plo75], call-by-need semantics [AFM$^+$95,AF97], state and control [FH92], and sharing and cycles [AK97,AB97]. Our notion of confluence with respect to evaluation avoids cyclic substitutions in the operational semantics, and so is related to the acyclic substitution restriction of Ariola and Klop [AK97].

This work is part of a renewed interest in linking issues that was inspired by Cardelli's call to arms [Car97]. Recent work on module systems and linking has focused on such issues as: sophisticated type systems for modules [HL94,Ler94]; the expressiveness of modules systems (e.g., handling features like recursive modules [FF98,CHP97,AZ99], inheritance and mixins [DS96,AZ99] and dynamic linking [FF98,WV99]); binary compatibility in the context of program modifications [SA93,DEW99]; and modularizing module systems [Ler96,AZ99]. There has been relatively little focus on issues related to link-time optimization; exceptions are [Fer95] and recent work on just-in-time compilers (e.g, [PC97]).

Our work stands out from other work on modules in two important respects. First, we partition the reduction relation of the calculus ($\rightarrow$) into *evaluation* (sometimes called *standard*) steps ($\Rightarrow$) that define a small-step operational semantics and *non-evaluation* (*non-standard*) steps ($\hookrightarrow$). While this partitioning is common in the calculus world (e.g., [Plo75,FH92,AF97]), it is rare in the module

[1] We will often abbreviate the name of this property as "soundness".

world. Typical work on modules (e.g., [Car97,AZ99]) gives only an operational semantics for modules. Yet in the context of link-time compilation, the notion of reduction in a calculus is essential for justifying meaning preserving program transformations. Without non-evaluation steps, even simple transformations like transforming $[F \mapsto \lambda x.(1+2)]$ to $[F \mapsto \lambda x.3]$ or $[A \mapsto 4, F \mapsto \lambda x.x + A]$ to $[A \mapsto 4, F \mapsto \lambda x.x + 4]$ are difficult to prove meaning preserving.

Second, unlike most recent work on modules (with the notable exception of [WV99]), our work considers only an untyped module language. There are several reasons for this. First, types are orthogonal to our focus on computational soundness and weak distributivity; types would only complicate the presentation. Second, introducing types often requires imposing restrictions that we would like to avoid. For example, to add types to their system, [AZ99] need to impose several restrictions on their untyped language: no components with recursive types, and no modules as components to other modules. Finally, we do not yet have anything new to say in the type dimension. We believe that it is straightforward to adapt an existing simple module type system (e.g., [Car97,FF98,AZ99]) to our calculus. On the other hand, we think that enriching our module system with polymorphic types is a very interesting avenue for future exploration.

Due to space limitations, our presentation is necessarily dense and telegraphic. Please see the companion technical report [MT00] for a more detailed exposition with additional explanatory text, more examples, and proofs.

2 The Module Calculus

In this section, we present a stratified calculus with three levels: a term calculus $\mathcal{T}$, a core module calculus $\mathcal{C}$, and a full module calculus $\mathcal{F}$. The three calculi are summarized in Fig. 1. Let $\mathcal{X}$ range over $\{\mathcal{T}, \mathcal{C}, \mathcal{F}\}$. The definition for each calculus $\mathcal{X}$ consists of the following:

- The syntax for calculus terms $\mathbf{Term}_{\mathcal{X}}$ and for general one-hole contexts $\mathbf{Context}_{\mathcal{X}}$. If $\mathbb{X} \in \mathbf{Context}_{\mathcal{X}}$, then $\mathbb{X}\{Y\}$ denotes the result of filling the hole of $\mathbb{X}$ with a term Y. Due to the hierarchical structure of our module calculus, Y is not necessarily a term of $\mathcal{X}$. For instance, in our hierarchy $\mathcal{T}$ contexts are filled with $\mathcal{T}$ terms; $\mathcal{C}$ contexts are filled with $\mathcal{T}$ terms; and $\mathcal{F}$ contexts are filled with either $\mathcal{C}$ or $\mathcal{F}$ terms. We assume that the notation $\mathbb{X}\{Y\}$ is only applied to such $\mathbb{X}$ and Y that the result of the filling is a well-formed term in $\mathbf{Term}_{\mathcal{X}}$. For instance, the notation $\mathbb{D}\{M\}$ is defined only if the resulting module is well-defined element of $\mathbf{Term}_{\mathcal{C}}$.
- A small-step operational semantics of $\mathcal{X}$ defined via an evaluation step relation $\Rightarrow_{\mathcal{X}}$, and a complementary definition of a non-evaluation step relation $\hookrightarrow_{\mathcal{X}}$. For each of the three calculi we define a one-step calculus relation $\rightarrow_{\mathcal{X}} \stackrel{\text{def}}{=} \Rightarrow_{\mathcal{X}} \cup \hookrightarrow_{\mathcal{X}}$.[2] The relation $\Rightarrow_{\mathcal{X}}$ is often defined in terms of an *evaluation context* $\mathbf{EvalContext}_{\mathcal{X}} \subseteq \mathbf{Context}_{\mathcal{X}}$. A term Y is a $\rightarrow_{\mathcal{X}}$*-normal-form*

[2] Alternatively we could have defined the rules for $\rightarrow_{\mathcal{X}}$ explicitly and then set $\hookrightarrow_{\mathcal{X}}$ to be $\rightarrow_{\mathcal{X}} \setminus \Rightarrow_{\mathcal{X}}$. However, giving explicit rules for $\hookrightarrow_{\mathcal{X}}$ clarifies the presentation.

Syntax for the Term Calculus ($\mathcal{T}$):

$c \in \mathbf{Const}$ = *constant values* $\quad x \in \mathbf{Variable}$ = *term variables*
$v \in \mathbf{Visible}$ = *external labels* $\quad h \in \mathbf{Hidden}$ = *internal labels*
$k, l \in \mathbf{Label} = \mathbf{Visible} \cup \mathbf{Hidden}$
$L, M, N \in \mathbf{Term}_{\mathcal{T}} ::= c \mid x \mid l \mid (\lambda x.M) \mid M_1 \mathbin{@} M_2 \mid M_1 \mathbin{\hat{\mathrm{op}}} M_2$
$\mathbb{C} \in \mathbf{Context}_{\mathcal{T}} ::= \Box \mid (\lambda x.\mathbb{C}) \mid \mathbb{C} \mathbin{@} M \mid M \mathbin{@} \mathbb{C} \mid \mathbb{C} \mathbin{\hat{\mathrm{op}}} M \mid M \mathbin{\hat{\mathrm{op}}} \mathbb{C}$
$V \in \mathbf{Value}_{\mathcal{T}} ::= c \mid x \mid \lambda x.M$

Notion of Reduction on Terms:

$$(\lambda x.M \mathbin{@} V) \rightsquigarrow_{\mathcal{T}} M[x := V] \qquad (\beta)$$
$$c_1 \mathbin{\hat{\mathrm{op}}} c_2 \rightsquigarrow_{\mathcal{T}} c, \text{where } c = \delta(\hat{\mathrm{op}}, c_1, c_2) \quad (\delta)$$

Evaluation and Non-evaluation Steps:

$\mathbb{E} \in \mathbf{EvalContext}_{\mathcal{T}} ::= \Box \mid \mathbb{E} \mathbin{@} M \mid (\lambda x.M) \mathbin{@} \mathbb{E} \mid \mathbb{E} \mathbin{\hat{\mathrm{op}}} M \mid c \mathbin{\hat{\mathrm{op}}} \mathbb{E}$

$$\mathbb{E}\{R\} \Rightarrow_{\mathcal{T}} \mathbb{E}\{Q\}, \quad \text{where} \quad R \rightsquigarrow_{\mathcal{T}} Q, \qquad \text{(term-ev)}$$
$$\overline{\mathbb{E}}\{R\} \hookrightarrow_{\mathcal{T}} \overline{\mathbb{E}}\{Q\}, \quad \text{where} \quad R \rightsquigarrow_{\mathcal{T}} Q. \qquad \text{(term-nev)}$$

Syntax for the Core Module Calculus ($\mathcal{C}$):

$D \in \mathbf{Term}_{\mathcal{C}} ::= [l_1 \mapsto M_1, \ldots, l_n \mapsto M_n]$ (abbreviated $[l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$),
provided $l_i = l_j$ implies $i = j$, $FV(D) = \emptyset$, and $\mathit{Imports}(D) \cap \mathbf{Hidden} = \emptyset$.

$\mathbb{D} \in \mathbf{Context}_{\mathcal{C}} ::= [l_i \overset{k-1}{\underset{i=1}{\mapsto}} M_i, l_k \mapsto \mathbb{C}, l_j \overset{n}{\underset{j=k+1}{\mapsto}} M_j]$

Projection Notation: $[l_i \overset{n}{\underset{i=1}{\mapsto}} M_i] \downarrow l_j = M_j$, if $1 \le j \le n$, and otherwise undefined.

Evaluation and Non-evaluation Steps:

$\mathbb{G} \in \mathbf{EvalContext}_{\mathcal{C}} ::= [l_i \overset{k-1}{\underset{i=1}{\mapsto}} M_i, l_k = \mathbb{E}, l_j \overset{n}{\underset{j=k+1}{\mapsto}} M_j]$

$$\mathbb{G}\{R\} \Rightarrow_{\mathcal{C}} \mathbb{G}\{Q\}, \text{ where } R \rightsquigarrow_{\mathcal{T}} Q. \qquad \text{(comp-ev)}$$
$$\mathbb{G}\{l\} \Rightarrow_{\mathcal{C}} \mathbb{G}\{V\}, \text{ where } \mathbb{G}\{l\} \downarrow l = V. \qquad \text{(subst-ev)}$$
$$[l_i \overset{n}{\underset{i=1}{\mapsto}} M_i, h_j \overset{m}{\underset{j=1}{\mapsto}} V_j] \Rightarrow_{\mathcal{C}} [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i], \text{ where } \forall_{1 \le i \le m}.h_i \notin \cup_{j=1}^{n} FL(M_j) \quad \text{(GC)}$$
$$\overline{\mathbb{G}}\{R\} \hookrightarrow_{\mathcal{C}} \overline{\mathbb{G}}\{Q\}, \text{ where } R \rightsquigarrow_{\mathcal{T}} Q. \qquad \text{(comp-nev)}$$
$$\overline{\mathbb{G}}\{l\} \hookrightarrow_{\mathcal{C}} \overline{\mathbb{G}}\{V\}, \text{ where } \overline{\mathbb{G}}\{l\} \downarrow l = V. \qquad \text{(subst-nev)}$$

Syntax for the Full Module Calculus ($\mathcal{F}$):

$F \in \mathbf{Term}_{\mathcal{F}} ::= D \mid I \mid F_1 \oplus F_2 \mid F[l \leftarrow l'] \mid \mathbf{let}\ I = F_1\ \mathbf{in}\ F_2$
$\mathbb{F} \in \mathbf{Context}_{\mathcal{F}} ::= \Box \mid \mathbb{F} \oplus F \mid F \oplus \mathbb{F} \mid \mathbb{F}[l \leftarrow l'] \mid \mathbf{let}\ I = \mathbb{F}\ \mathbf{in}\ F \mid \mathbf{let}\ I = F\ \mathbf{in}\ \mathbb{F}$

Evaluation and Non-evaluation Steps:

$$D \Rightarrow_{\mathcal{F}} D', \text{ where } D \Rightarrow_{\mathcal{C}} D' \qquad \text{(mod-ev)}$$
$$\mathbb{F}\{[k_i \overset{n}{\underset{i=1}{\mapsto}} M_i] \oplus [l_j \overset{m}{\underset{j=1}{\mapsto}} N_j]\} \Rightarrow_{\mathcal{F}} \mathbb{F}\{[k_i \overset{n}{\underset{i=1}{\mapsto}} M_i, l_j \overset{m}{\underset{j=1}{\mapsto}} N_j]\}, \qquad \text{(link)}$$
$$\text{where } (\cup_{i=1}^{n} k_i) \cap (\cup_{j=1}^{m} l_j) = \emptyset$$
$$\mathbb{F}\{D[l \leftarrow k]\} \Rightarrow_{\mathcal{F}} \mathbb{F}\{D[l := k]\}, \qquad \text{(rename)}$$
$$\text{where } l \in BL(D) \text{ implies } k \notin BL(D),$$
$$l \in \mathbf{Hidden} \text{ implies } k \in \mathbf{Hidden}, \text{ and}$$
$$k \in \mathbf{Hidden} \text{ implies } l \notin \mathit{Imports}(D).$$
$$\mathbb{F}\{\mathbf{let}\ I = F_1\ \mathbf{in}\ F_2\} \Rightarrow_{\mathcal{F}} \mathbb{F}\{F_1[I := F_2]\}, \qquad \text{(let)}$$
$$\mathbb{F}\{D\} \hookrightarrow_{\mathcal{F}} \mathbb{F}\{D'\}, \text{ where } D \rightarrow_{\mathcal{C}} D' \qquad \text{(mod-nev)}$$
$$\text{and } \mathbb{F} \neq \Box \text{ or } D \hookrightarrow_{\mathcal{C}} D'$$

Fig. 1. The three levels of the module calculus.

(NF) if there is no term N s.t. $M \rightarrow_{\mathcal{X}} N$, a $\Rightarrow_{\mathcal{X}}$*-NF* is defined analogously. For each calculus $\mathcal{X}$, there is a classification function $Cl_{\mathcal{X}}$ that maps each term to a "class" token that describes its state w.r.t. evaluation. The classes for evaluatable terms must be disjoint from those in $\Rightarrow_{\mathcal{X}}$-NF. Also associated with each calculus $\mathcal{X}$ is a set $\mathbf{Value}_{\mathcal{X}}$ of *values* that is the union of one or more classes of $\Rightarrow_{\mathcal{X}}$-NFs. The function $Outcome_{\mathcal{X}}$ of a term is defined to be the class of its $\Rightarrow$-normal form or a symbol $\perp$ if the term diverges.

We use the following notations and conventions. If $\mathbb{X}$ ranges over $\mathbf{EvalContext}_{\mathcal{X}}$, then $\overline{\mathbb{X}}$ ranges over $\mathbf{Context}_{\mathcal{X}} \setminus \mathbf{EvalContext}_{\mathcal{X}}$ (i.e. the set of *non-evaluation contexts*). For pairs of rules such as (comp-ev) and (comp-nev), which only differ by the use of an evaluation versus a non-evaluation context, we introduce a notation for the combined calculus rule. For instance, we say that $D \rightarrow_{\mathcal{C}} D'$ by the rule (comp) if either $D \Rightarrow_{\mathcal{C}} D'$ by (comp-ev) or $D \hookrightarrow_{\mathcal{C}} D'$ by (comp-nev). If $\rightarrow$ is a one-step relation, then $\rightarrow^*$ denotes its reflexive transitive closure, and $\leftrightarrow$ denotes its reflexive, symmetric, and transitive closure.

The following properties of calculi are important in the sequel.

Definition 1 (Confluence). *The $\rightarrow$ relation is confluent if $M_1 \rightarrow^* M_2$ and $M_1 \rightarrow^* M_3$ implies the existence of M_4 s.t. $M_2 \rightarrow^* M_4$, $M_3 \rightarrow^* M_4$. A calculus X has confluence if $\rightarrow_{\mathcal{X}}$ is confluent.*

Definition 2 (Standardization). *A calculus $\mathcal{X}$ has the standardization property if for any sequence $M_1 \rightarrow^*_{\mathcal{X}} M_2$ there exists M_3 s.t. $M_1 \Rightarrow^*_{\mathcal{X}} M_3 \hookrightarrow^*_{\mathcal{X}} M_2$.*

2.1 Term calculus ($\mathcal{T}$)

The module calculus is built on top of a term calculus $\mathcal{T}$, a typical call-by-value λ-calculus that includes constants (which we assume include integers) and binary operators (we assume ôp includes standard integer operations). For interfacing with the module language in which it is embedded, the term syntax also includes two disjoint classes of labels whose union, **Label**, is itself disjoint from **Variable**.

We adopt the convention that all λ-bound variables in a term must be distinct. The free variables of a term M, written $FV(M)$, are defined as usual (recall that variables are distinct from labels). The set of labels appearing in a term M is written $FL(M)$; because labels cannot be λ-bound, they always appear "free". The result of a capture-avoiding substitution of M' for x in M is written $M[x := M']$. In addition to using α-renaming to avoid variable capture during substitution, it may be necessary to α-rename the result of substitution to maintain the distinct variable naming invariant. The result of substituting a term M' for a label l in M is written $M[l := M']$.

Both $\Rightarrow_{\mathcal{T}}$ and $\hookrightarrow_{\mathcal{T}}$ are defined via a redex/contractum relation $\rightsquigarrow_{\mathcal{T}}$ specified by a call-by-value β rule and a δ rule (unspecified) for binary functions on constants. Terms in $dom(\rightsquigarrow_{\mathcal{T}})$ are called *term redexes*. The relations $\Rightarrow_{\mathcal{T}}$ and $\hookrightarrow_{\mathcal{T}}$ are contextual closures of $\rightsquigarrow_{\mathcal{T}}$ with respect to an *evaluation context* $\mathbb{E}$ and a *non-evaluation context* $\overline{\mathbb{E}}$. It is easy to see that $\rightarrow_{\mathcal{T}}$ (defined as $\Rightarrow_{\mathcal{T}} \cup \hookrightarrow_{\mathcal{T}}$) is the contextual closure of $\rightsquigarrow_{\mathcal{T}}$ with respect to a general context $\mathbb{C}$.

A term M can be uniquely classified with respect to evaluation via $Cl_{\mathcal{T}}(M)$, defined as:

const(c)	if $M = c$	**abs**	if $M = \lambda x.N$	**evaluatable**	if $M = \mathbb{E}\{R\}$
var	if $M = x$	**stuck**(l)	if $M = \mathbb{E}\{l\}$	**error**	otherwise

It turns out that an evaluatable term M can be uniquely parsed into $\mathbb{E}$ and R such that $M = \mathbb{E}\{R\}$, so $\Rightarrow_{\mathcal{T}}$ is deterministic (i.e., it is a partial function rather than a relation). The partial function $Eval_{\mathcal{T}}(M)$ is defined as the $\Rightarrow_{\mathcal{T}}$-NF of M if it exists; otherwise, M is said to diverge. The total function $Outcome_{\mathcal{T}}(M)$ is defined as $Cl_{\mathcal{T}}(Eval_{\mathcal{T}}(M))$ if $Eval_{\mathcal{T}}(M)$ is defined, and $\perp$ if M diverges. Using classical techniques [Plo75,Bar84], it is straightforward to prove that $\rightarrow_{\mathcal{T}}$ is confluent, and $\mathcal{T}$ has the standardization property.

2.2 Core Module Calculus ($\mathcal{C}$)

In our module calculus, modules are unordered collections of labeled terms. There are two disjoint classes of labels: *visible* and *hidden*. Visible labels name components to be exported to other modules, and also name import sites within a component, while hidden labels name components that can only be referenced within the module itself. (This distinction is similar to distinction between deferred variables and expression names on one hand and local variables on the other in [AZ99]). Intuitively, a module is a fragment of a recursively scoped record that can be dynamically constructed by linking, where visible labels serve to "wire" the definitions in one module to the uses in another.

A *module binding* is written $l \mapsto M$. A *module* is a bracketed set of such bindings in which the labels of any two bindings are distinct. Note that a hole in a module context $\mathbb{D}$ is filled with a $\mathcal{T}$-term rather than another module. The notation $l_i \overset{n}{\underset{i=1}{\mapsto}} M_i$ stands for the bindings $l_1 \mapsto M_1 \ldots l_n \mapsto M_n$, and $D \downarrow l$ extracts the component M bound to l in D (if it exists).

Suppose that $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$. The free variables of D are $FV(D) = \cup_{i=1}^{n} FV(M_i)$. The substitution $D[l := k]$ yields $[l'_i \overset{n}{\underset{i=1}{\mapsto}} M_i[l := k]]$, where $l'_i = k$ if $l_i = l$ and $l'_i = l_i$ otherwise. The set of *bound labels* in D is defined as $BL(D) = \cup_{i=1}^{n} l_i$, while the set of *free labels* is $FL(D) = (\cup_{i=1}^{n} FL(M_i))/BL(D)$. The *exported labels* of D are those that are both bound and visible ($Exports(D) = BL(D) \cap \mathbf{Visible}$), while the *imported labels* are just the free ones ($Imports(D) = FL(D)$). In order to be well-formed, a module D must satisfy three conditions: (1) all its bound labels must be distinct; (2) it must not import any hidden labels; and (3) it must not contain any free variables (such variables would necessarily be unbound). In a well-formed module, the hidden labels are necessarily bound, so we define $Hid(D) = BL(D) \cap \mathbf{Hidden}$.

The evaluation relation $\Rightarrow_{\mathcal{C}}$ is defined using a *module evaluation context* $\mathbb{G}$ which lifts term-level evaluation context $\mathbb{E}$ to the module level. The three rules of $\Rightarrow_{\mathcal{C}}$ allow the following reductions: (comp-ev) lifts $\Rightarrow_{\mathcal{T}}$ to the module level; (subst-ev) substitutes a labeled value for a label occurrence in the module; (GC)

garbage collects hidden values not referenced elsewhere in the module. Unlike $\Rightarrow_{\mathcal{T}}$, $\Rightarrow_{\mathcal{C}}$ is not deterministic, because it can perform an evaluation step on any component. Nevertheless, $\Rightarrow_{\mathcal{C}}$ is confluent. The complementary relation $\hookrightarrow_{\mathcal{C}}$ has two rules (comp-nev) and (subst-nev) which differ from their evaluation analogs by using a non-evaluation context in place of an evaluation context. Note that the (GC) rule does not have a non-evaluation counterpart; i.e., all (GC)-reductions are evaluation steps.

Let us consider some examples of module reductions.[3] Any one-step reduction on a term component can be lifted to the module via the (comp) rule: $[F \mapsto \lambda x.1 + 2] \hookrightarrow_{\mathcal{C}} [F \mapsto =\lambda x.3]$. This is a non-evaluation step, since the redex occurs under a λ. As an example of (subst), consider $[A \mapsto 4, F \mapsto A + 3] \Rightarrow_{\mathcal{C}} [A \mapsto 4, F \mapsto 4 + 3]$. Here A in the second term appears in an evaluation context. Note that a value may be substituted into itself: $[F \mapsto \lambda x.F] \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.(\lambda x_1.F)] \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.(\lambda x_1.(\lambda x_2.(\lambda x_3.F)))]$ (where α-renaming preserves the distinct variable invariant). This is a non-evaluation step, since F appears under a λ. The (GC) rule garbage collects hidden values not referenced elsewhere in the module. Consider:

$$[P \mapsto \lambda w.g \mathbin{@} (w + 1), f \mapsto \lambda x.h, g \mapsto \lambda y.y * 2, h \mapsto \lambda z.f] \\ \Rightarrow_{\mathcal{C}} [P \mapsto \lambda w.g \mathbin{@} (w + 1), g \mapsto \lambda y.y * 2]$$

The mutually recursive bindings for f and h can be removed because all references to these hidden labels occur inside of the values named by these labels. However, g cannot be removed, since an exported term references it.

It turns out that $\mathcal{C}$ has the standardization property. But interestingly, even though $\Rightarrow_{\mathcal{C}}$ is confluent, $\rightarrow_{\mathcal{C}}$ is *not* confluent, due to the possibility of mutually recursive (subst) redexes that appear under a λ and therefore not in an evaluation context. Consider an example due to [AK97]: $D_0 = [F \mapsto \lambda x.G, G \mapsto \lambda y.F]$. Then $D_0 \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.\lambda y'.F, G \mapsto \lambda y.F] = D_1$ and $D_0 \hookrightarrow_{\mathcal{C}} [F \mapsto \lambda x.G, G \mapsto \lambda y.\lambda x'.G] = D_2$. D_1 (resp. D_2) has an even (resp. odd) number of λs for F and an odd (resp. even) number for G, and in every reduction sequence starting with D_1 (resp. D_2), all terms will have this property. Clearly, reduction sequences starting at D_1 and D_2 can never meet at a common term.

The confluence of $\Rightarrow_{\mathcal{C}}$ gives rise to a partial function $\mathit{Eval}_{\mathcal{C}}(D)$ that, when defined, returns a module whose components are all $\Rightarrow_{\mathcal{T}}$-normal forms. The classification notion also lifts to the module level: $\mathit{Cl}_{\mathcal{C}}(D) = [l_i \overset{n}{\underset{i=1}{\mapsto}} \mathit{Cl}_{\mathcal{T}}(M_i)]$, where $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$. As in the term calculus, $\mathit{Outcome}_{\mathcal{C}}(D) = \mathit{Cl}_{\mathcal{C}}(\mathit{Eval}_{\mathcal{C}}(D))$ if $\mathit{Eval}_{\mathcal{C}}(D)$ exists, and $\bot$ otherwise. We say that $D = [l_i \overset{n}{\underset{i=1}{\mapsto}} V_i]$ is a *module value* ($D \in \mathbf{Value}_{\mathcal{C}}$) if $\mathit{Hid}(D) = \emptyset$ and $V_i \in \mathbf{Value}_{\mathcal{T}}$ for all $1 \leq i \leq n$.

2.3 Full Module Calculus ($\mathcal{F}$)

The full module calculus extends the core module calculus with three module operators: linking, renaming, and binding. Intuitively, the linking of modules D_1

[3] In examples, we adopt the convention that visible labels have uppercase names while hidden labels have lowercase names.

and D_2, written $D_1 \oplus D_2$, takes the union of their bindings. To avoid naming conflicts between both visible and hidden labels, $BL(D_1)$ and $BL(D_2)$ must be disjoint. The fact that the import labels of a well-formed module may not be hidden prevents the components of one module from accessing hidden components of another when they are linked.

The renaming operator renames any module label (visible or hidden, import or export). Renaming import and export labels is the way to connect an exported component of one module to an import site in another. Renaming a visible label to a fresh hidden label hides a component; a user-level "hiding" operator could be provided as syntactic sugar for such renaming. Finally, renaming of hidden variables to other hidden variables is necessary to guarantee that hidden variables are disjoint when they are linked. The side conditions on renaming prevents certain undesirable scenarios: (1) attempting to rename one bound label to another (causing a name clash); (2) renaming a hidden variable to a visible one, thereby exposing it; and (3) renaming a (necessarily visible) import to a hidden label, thereby making the module ill-formed.

The binding operator **let** $I = F_1$ **in** F_2 names the (result of evaluating the) definition term F_1 and uses the name within the body term F_2. This models situations in which the same module is used multiple times in different contexts.

The disjoint hidden label requirement for $\oplus$ simplifies reasoning about the calculus, but is severe from the perspective of a user, who should not be able to predict the names of the hidden labels of any module. We address this problem by supplying a user-level linking operator $\overline{\oplus}$ that can be defined in terms of the primitive linking operator $\oplus$ and renaming, as follows. Suppose that $F_1 = [v_i \overset{n_1}{\underset{i=1}{\mapsto}} M_i, h_j \overset{m_1}{\underset{j=1}{\mapsto}} N_j]$ and $F_2 = [v'_i \overset{m_2}{\underset{i=1}{\mapsto}} M'_i, h'_j \overset{n_2}{\underset{j=1}{\mapsto}} N'_j]$. Then $F_1 \overline{\oplus} F_2$ is defined as:

$$F_1[h_1 \leftarrow h''_1, \ldots, h_{n_1} \leftarrow {h_{n_1}}''] \ \oplus \ F_2[h'_1 \leftarrow {h_{n_1+1}}'', \ldots, h'_{n_2} \leftarrow {h_{n_1+n_2}}''],$$
$$\text{where } \left((\bigcup_{i=1}^{n_1} h_i) \cup (\bigcup_{j=1}^{n_2} {h_j}')\right) \bigcap \left(\bigcup_{k=1}^{n_1+n_2} {h_k}''\right) = \emptyset$$

The hidden labels of F_1 and F_2 are renamed to fresh hidden labels before the modules are linked to avoid collisions. The renaming performed by $\overline{\oplus}$ is is similar to the α-renaming required in other module calculi linking operations (e.g., in [FF98] when rewriting the **compound** linking form to the **unit** module form).

The definition of $\rightarrow_{\mathcal{F}}$ lifts core module reduction steps to the module expression level and adds evaluation rules for the link-level operators (link, rename, and bind). The structure of $\mathbf{Context}_{\mathcal{F}}$ allows the link-level operators to be evaluated in any order. The lifted core module reduction steps are only considered evaluation steps if they are not surrounded by any link-level operators; this forces all link-level steps to be performed first in a "link-time stage", followed by a "run-time stage" of core module steps.

The lack of confluence of $\rightarrow_{\mathcal{C}}$ is inherited by $\rightarrow_{\mathcal{F}}$, but we are still able to show that $\Rightarrow_{\mathcal{F}}$ is confluent and $\mathcal{F}$ has the standardization property. If F is a link, rename, or bind term, we define $Cl_{\mathcal{F}}(F)$ to be **linkable**; otherwise we define $Cl_{\mathcal{F}}(F)$ to be $Cl_{\mathcal{C}}(F)$ (in this case, $F \in \mathbf{Term}_{\mathcal{C}}$). $Outcome_{\mathcal{F}}$ is defined analogously with $Outcome_{\mathcal{C}}$, and $\mathbf{Value}_{\mathcal{F}} = \mathbf{Value}_{\mathcal{C}}$.

3 Meaning Preservation

The calculus defined in the previous section allows us to reason about module transformations. A *transformation* T of a calculus $\mathcal{X}$ is a relation $T : \mathcal{X} \times \mathcal{X}$. Even though T in general is not a function, we sometimes write $Z = T(Y)$ if $(Y, Z) \in T$. Below we define a notion of observational equivalence and, based on it, a notion of a meaning preserving transformation.

Definition 3 (Observational Equivalence). *Two terms Y and Z of a calculus $\mathcal{X}'$ are observationally equivalent in a calculus $\mathcal{X}$ (written $Y \cong_{\mathcal{X}} Z$) if for all contexts $\mathbb{X}$ s.t. $\mathbb{X}\{Y\}$ and $\mathbb{X}\{Z\}$ are well-formed terms of $\mathcal{X}$, $\mathbb{X}\{Y\} \Rightarrow^*_{\mathcal{X}} W$ iff $\mathbb{X}\{Z\} \Rightarrow^*_{\mathcal{X}} W'$ where W and $W' \in \mathbf{Value}_{\mathcal{X}}$ and $Cl_{\mathcal{X}}(W) = Cl_{\mathcal{X}}(W')$.*

In the definition, note that $\mathcal{X}$ may or may not be the same as $\mathcal{X}'$. As an example, two core modules are observationally equivalent in $\mathcal{F}$ if in any full module context $\mathbb{F}$ they evaluate to module values of the same class, as defined above. For instance, consider the following modules: $D_1 = [F \mapsto \lambda x.x + a, a \mapsto 1 + 2]$, $D'_1 = [F \mapsto \lambda x.x + 3, a \mapsto 3]$, $D_2 = [S \mapsto N_1 + N_2]$, and $D'_2 = [S \mapsto N_2 + N_1]$. $D_1 \cong_{\mathcal{F}} D'_1$ because the exported F behaves like an "add 3" function for both modules in any context. Assuming that + is commutative, $D_2 \cong_{\mathcal{F}} D'_2$ because they evaluate to the same module value when they are placed in a context that supplies integer values for N_1 and N_2, and none of the two modules evaluates to a module value if the context does not supply such values.

Definition 4 (Meaning Preservation). *A transformation T of a calculus X' is meaning preserving in a calculus $\mathcal{X}$ if $(Y, Z) \in T$ implies $Y \cong_{\mathcal{X}} Z$.*

For instance, the constant folding/propagation transformation CFP in $\mathcal{C}$ is meaning preserving in $\mathcal{F}$, as seen in the above example with D_1 and D'_1. The example with D_2 and D'_2 illustrates that a transformation SPO that swaps the operands of + in $\mathcal{C}$ is also meaning preserving in $\mathcal{F}$.

3.1 Computational Soundness

Proving that a transformation is meaning preserving can be difficult and tedious work. However, if T is a *calculus-based transformation* in $\mathcal{X}$, i.e. $Y \leftrightarrow_{\mathcal{X}} Z$ for all $(Y, Z) \in T$, then it is automatically meaning preserving in a calculus $\mathcal{X}'$ satisfying the conditions of Lemma 1 below.

A key notion for showing the meaning preservation of calculus-based transformations is computational soundness:

Definition 5 (Computational Soundness). *A calculus $\mathcal{X}$ is computationally sound if $M \leftrightarrow_{\mathcal{X}} N$ implies $Outcome_{\mathcal{X}}(M) = Outcome_{\mathcal{X}}(N)$, where $M, N \in \mathbf{Term}_{\mathcal{X}}$.*

It follows from computational soundness that if two terms are equivalent in the calculus then they are observationally equivalent in an *empty* context. For observationally equivalence to hold in *all* contexts requires *embedding*:

Definition 6 (Embedding). *A relation $\rightarrow_{\mathcal{X}'}$ is embedded in a relation $\rightarrow_{\mathcal{X}}$ (written $\rightarrow_{\mathcal{X}'} \preceq \rightarrow_{\mathcal{X}}$) if $Y \rightarrow_{\mathcal{X}'} Z$ implies that $\mathbb{X}\{Y\} \rightarrow_{\mathcal{X}} \mathbb{X}\{Z\}$ for any context $\mathbb{X}$ s.t. $\mathbb{X}\{Y\}$ and $\mathbb{X}\{Z\}$ are well-formed terms of $\mathcal{X}$.*

As examples of embeddings, in our module calculus, $\rightarrow_{\mathcal{T}} \preceq \rightarrow_{\mathcal{C}}$ (because term reductions can be performed in the bindings of a module) and $\rightarrow_{\mathcal{C}} \preceq \rightarrow_{\mathcal{F}}$ (because core module reductions can be performed within a full module term). The self-embedding $\rightarrow_{\mathcal{X}} \preceq \rightarrow_{\mathcal{X}}$ means that the relation $\rightarrow_{\mathcal{X}}$ is a congruence relative to the one-holed contexts of $\mathcal{X}$. For instance, $\rightarrow_{\mathcal{T}}$ and $\rightarrow_{\mathcal{F}}$ are both congruences since they are embedded in themselves.

Together, computational soundness and embedding imply that calculus-based transformations are meaning preserving.

Lemma 1. *If a calculus $\mathcal{X}$ is sound and $\rightarrow_{\mathcal{X}'} \preceq \rightarrow_{\mathcal{X}}$, then any calculus-based transformation T in $\mathcal{X}'$ is meaning preserving in $\mathcal{X}$.*

Proof. By Definition 6, $Y \leftrightarrow_{\mathcal{X}'} T(Y)$ implies that for any context $\mathbb{X}$, $\mathbb{X}\{Y\} \leftrightarrow_{\mathcal{X}} \mathbb{X}\{T(Y)\}$. Then $Outcome_{\mathcal{X}}(\mathbb{X}\{Y\}) = Outcome_{\mathcal{X}}(\mathbb{X}\{T(Y)\})$ by soundness of $\mathcal{X}$. By the definition of $Outcome_{\mathcal{X}}$, $\mathbb{X}\{Y\} \Rightarrow^*_{\mathcal{X}} W$ iff $\mathbb{X}\{T(Y)\} \Rightarrow^*_{\mathcal{X}} W'$, where W and $W' \in \Rightarrow_{\mathcal{X}}$-NF and $Cl_{\mathcal{X}}(W) = Cl_{\mathcal{X}}(W')$. Since $\mathbf{Value}_{\mathcal{X}}$ respects the ordering of $Cl_{\mathcal{X}}$, W and W' are either both in or both not in $\mathbf{Value}_{\mathcal{X}}$. □

The soundness of the call-by-name and call-by-value λ-calculi are a classic result due to Plotkin [Plo75]. Since the reduction relations of these caculi are congruences (i.e., are self-embedded), Lemma 1 implies that all calculus-based transformations in these calculi are meaning preserving.

A main result of our work is that $\mathcal{T}$, $\mathcal{C}$, and $\mathcal{F}$ are all computationally sound. Given the four embeddings for these calculi enumerated above, Lemma 1 implies that calculus-based transformations are meaning preserving in each of the four cases. Many classic program transformations (both at the term and at the module level) fall into this category: e.g., constant folding and propagation, function inlining, and simple forms of dead-code elimination that eliminate unused value bindings. All of these (and any combinations thereof) can easily be shown to be meaning preserving because all are justified by simple calculus steps.

We emphasize that there are numerous common transformations that are *not* calculus-based and so their meaning preservation cannot be shown via this technique. The operand-swapping SPO transformation introduced above is in this category. Note that $Outcome_{\mathcal{C}}(D_2) = [S \mapsto \mathbf{stuck}(N_1)]$ and $Outcome_{\mathcal{C}}(D'_2) = [S \mapsto \mathbf{stuck}(N_2)]$, underscoring that SPO cannot possibly be expressed via calculus steps. Global transformations like closure conversion, assignment conversion, uncurrying, etc., are other examples of non-calculus-based transformations.

3.2 A Novel Technique for Proving Soundness

As in Plotkin's approach, we show soundness of the module calculi in order to prove that calculus-based transformations are meaning preserving. However, we formulate and prove much more general conditions for soundness that do not

depend on the particulars of the module calculus or of the definition of a program outcome. We also extend traditionally used definitions to a hierarchy of calculi, allowing terms of one calculus to fill in contexts of another (see Definition 3 above). Our discussion is independent of the particulars of a calculus. The notations M, N for terms and $\mathbb{C}$ for contexts are used below for clarity (since these notations are more traditional); note that they are independent from the same notations used in the term calculus $\mathcal{T}$.

Traditional proofs of computational soundness depend on confluence of reduction in the calculus and on standardization, as well as on the following property, which is often not articulated, but plays a critical role in soundness proofs:

Definition 7 (Class Preservation). *Calculus $\mathcal{X}$ has the class preservation property if $M \hookrightarrow_{\mathcal{X}} N$ implies $Cl_{\mathcal{X}}(M) = Cl_{\mathcal{X}}(N)$, where $M, N \in \mathbf{Term}_{\mathcal{X}}$.*

Below we present a traditional proof of computational soundness that generalizes Plotkin's approach.

Theorem 1 (Soundness of a Confluent Calculus). *Confluence, standardization, and class preservation imply soundness.*

Proof. The diagram of the proof is shown in Fig. 2.[4] Assume that $M \leftrightarrow_{\mathcal{X}} N$ and that $M \Rightarrow^* M' = \mathit{Eval}(M)$. By confluence there exists L s.t. $M' \rightarrow^* L$, $N \rightarrow^* L$. Since M' is a normal form w.r.t. $\Rightarrow$, there can not be an evaluation sequence starting at M', so $M' \hookrightarrow^* L$. By standardization, $N \rightarrow^* L$ implies that there is N' s.t. $N \Rightarrow^* N' \hookrightarrow^* L$. Since M', L, and N' are connected only by $\hookrightarrow$, by class preservation, $\mathit{Cl}(M') = \mathit{Cl}(L) = \mathit{Cl}(N')$, and since N' is of the same class as M', it must also be a normal form w.r.t. $\Rightarrow$, so $N' = \mathit{Eval}(N)$.

Now assume that M diverges. If $\mathit{Eval}(N)$ exists, then by the above argument we can show that $\mathit{Eval}(M)$ exists as well. So if M diverges, then so does N. □

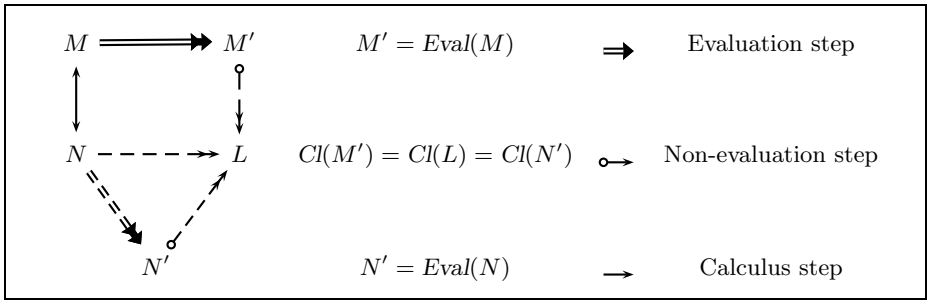

Fig. 2. Sketch of the traditional proof of computational soundness.

[4] In Figs. 2 and 3, double-headed arrows denote reflexive, transitive closures of the respective relations, and a line with arrows on both ends denotes the reflexive, symmetric, transitive closure of the respective relation.

The above approach does not work for a calculus that lacks confluence. But it turns out that general confluence is not required for soundness! Since the outcome of a term is defined via the evaluation reduction, we can instead use a weaker form of confluence: *confluence with respect to evaluation*. The two properties given below that we call *lift* and *project* (see also Fig. 3), together with the class preservation property, are sufficient to show soundness.

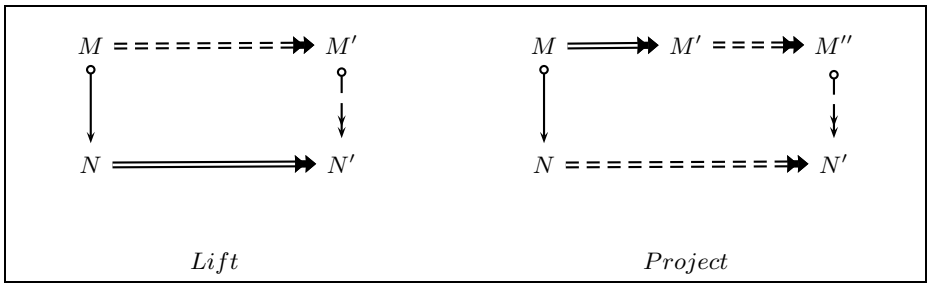

Fig. 3. The lift and project properties.

Definition 8 (Lift). *A calculus has the lift property if for any reduction sequence* $M \hookrightarrow N \Rightarrow^* N'$ *there exists a sequence* $M \Rightarrow^* M' \hookrightarrow^* N'$.

Definition 9 (Project). *A calculus has the project property if* $M \hookrightarrow N$, $M \Rightarrow^* M'$ *implies that there exist terms* M'', N' *s.t.* $M' \Rightarrow^* M''$, $N \Rightarrow^* N'$, *and* $M'' \hookrightarrow^* N'$.

The project property is the formalization of the notion of confluence w.r.t. evaluation mentioned above. It says that an evaluation step and a non-evaluation step leaving the same term can always be brought back together. The lift property is equivalent to standardization: any reduction sequence can be transformed into a standard sequence by pushing "backwards" sequences of evaluation steps through single non-evaluation steps. There is a benefit in proving standardization using the lift property (rather than directly): proofs of both the lift and project properties use the same mechanism (certain properties of residuals and finite developments [Bar84]) and share several intermediate results.

The following theorem embodies our new approach to proving soundness:

Theorem 2. *Suppose that* $\Rightarrow$ *is confluent. Then lift, project, and class preservation imply soundness.*

Proof. We want to show that if $M \leftrightarrow N$, then $\mathit{Outcome}(M) = \mathit{Outcome}(N)$. Without loss of generality assume that M and N are connected by a single step. Assume that $\mathit{Outcome}(M) \neq \bot$. Let $M' = \mathit{Eval}(M)$. In all four of the following cases, $\mathit{Outcome}(M) = \mathit{Outcome}(N)$:

– $M \hookrightarrow N$. By the project property, $M \Rightarrow^* M'$ implies that there exist M'', N' s.t. $M' \Rightarrow^* M''$, $N \Rightarrow^* N'$, and $M'' \hookrightarrow^* N'$. But M' is a normal form w.r.t. $\Rightarrow$, so $M' = M''$. By the class preservation property $Cl(M') = Cl(N')$, so N' is also a normal form. Hence $N' = Eval(N)$, and $Outcome(M) = Outcome(N)$.
– $N \hookrightarrow M$. Similar to the previous case by the lift property.
– $M \Rightarrow N$. By confluence of $\Rightarrow$ there exists N' s.t. $N \Rightarrow^* N'$, $M' \Rightarrow^* N'$. But M' is a normal form, so $N \Rightarrow^* M' = Eval(N)$.
– $N \Rightarrow M$. Then by transitivity of $\Rightarrow^*$, $N \Rightarrow^* M' = Eval(N)$.

Now let $Outcome(M) = \bot$. Assuming $Outcome(N) \neq \bot$, by the above argument $Outcome(M) = Outcome(N) \neq \bot$, and we get a contradiction. □

$\mathcal{C}$ and $\mathcal{F}$ satisfy the lift, project, and class preservation properties, so they enjoy the soundness property. For the technical details, consult [MT00].

4 Weak Distributivity

We say that a module transformation T is weakly distributive if and only if $T(D_1 \oplus D_2) = T(T(D_1) \oplus T(D_2))$, where $=$ is syntactic equality (modulo α-renaming and module binding order).

Let T_{link} be a single module transformation performing all link-time optimizations. Suppose that the translator from source modules to intermediate modules is given by $s2i(D) = T_{\text{link}}(D)$[5]. Also suppose that the linking operator on intermediate modules is defined as $D_1 \oplus_{\text{link}} D_2 = T_{\text{link}}(D_1 \oplus D_2)$. Then if T_{link} is weakly distributive, we have that $s2i(D_1) \oplus_{\text{link}} s2i(D_2) = T_{\text{link}}(T_{\text{link}}(D_1) \oplus T_{\text{link}}(D_2)) = T_{\text{link}}(D_1 \oplus D_2) = s2i(D_1 \oplus D_2)$. Thus, compiling a "link tree" of modules in the link-time compilation model gives exactly the same code as compilation in whole-program model. This is the sense in which weakly distributive transformations are promising candidates for link-time optimizations.

Here we briefly discuss two classes of weakly distributive module transformations T. We assume the following about T: (1) it is strongly normalizing; and (2), if T can be applied to a module $[X_i \overset{n}{\underset{i=1}{\mapsto}} M_i]$, then it can be applied to a module $[X_i \overset{n}{\underset{i=1}{\mapsto}} M_i, Y_j \overset{m}{\underset{j=1}{\mapsto}} N_j]$, i.e. to the same module with extra bindings. To motivate the second assumption, let FI be function inlining on modules restricted to non-recursive substitutions (so that the first assumption is satisfied). Consider the following inlining/linking sequence: $[X \mapsto \lambda w.Y, Z \mapsto \lambda x.X] \oplus [Y \mapsto \lambda y.Z] \overset{\text{FI}}{\rightarrow} [X \mapsto \lambda w.Y, Z \mapsto \lambda x.\lambda w'.Y] \oplus [Y \mapsto \lambda y.Z] \rightarrow_{\mathcal{F}} [X \mapsto \lambda w.Y, Z \mapsto \lambda x.\lambda w'.Y, Y \mapsto \lambda y.Z] \overset{\text{FI}}{\rightarrow} [X \mapsto \lambda w.\lambda y.Z, Z \mapsto \lambda x.\lambda w'.Y, Y \mapsto \lambda y.Z]$. On the other hand, linking first gives: $[X \mapsto \lambda w.Y, Z \mapsto \lambda x.X, Y \mapsto \lambda y.Z]$, and at this point the cycle becomes apparent, and no inlining is possible. Thus, extra bindings can prevent weak distributivity by blocking the transformation.

A simple class of weakly distributive transformations are those satisfying two conditions: (1) idempotence: $T(T(D)) = T(D)$; and (2) (strong) distributivity

[5] For simplicity, we assume the source and intermediate languages are the same.

over $\oplus$: $T(D_1 \oplus D_2) = T(D_1) \oplus T(D_2)$. It is easy to show that such a T is weakly distributive. Examples include many combinations of intra-term transformations, such as constant folding/propagation, dead code elimination, and function inlining (restricted to non-recursive cases). Note that the second condition implies that the transformation independently transforms the components of a module; i.e., the transformation cannot use the (subst) or (GC) rule.

Closures of confluent transformations T form another class of weakly distributive transformations. It is possible to simulate any transformation step in $T(T(D_1) \oplus T(D_2))$ by a corresponding step in $T(D_1 \oplus D_2)$. Using confluence, strong normalization, and the extra-bindings assumption, it can be shown that the two expressions transform to the same result. For example, constant folding/propagation at the module level (i.e., including the (subst) rule) has all of these properties, and so is weakly distributive.

5 Future Work

There are several directions in which we plan to extend the work presented here.

Types: We are exploring several type systems for our module calculus, especially ones which express polymorphism via intersection and union types. These have intriguing properties for modular analysis and link-time compilation [Jim96,Ban97,KW99].

Non-local Transformations: So far, we have only considered meaning preservation and weak distributivity in the context of simple local transformations. We are investigating global transformations like closure conversion, uncurrying, and useless variable elimination in the context of link-time compilation.

Weakening Weak Distributivity: Weak distributivity requires the rather strong condition of synactic equality between $T(D_1 \oplus D_2)$ and $T(T(D_1) \oplus T(D_2))$. Weaker notions of equality may also be suitable. Note that "has the same meaning as" is *too* weak, since it does not capture the pragmatic relationship between the two sides; they should have "about the same efficiency".

Abstracting over the Base Language: Our framework assumes that the module calculus is built upon a particular base calculus. Inspired by [AZ99], we would like to parameterize our module calculus over any base calculus.

Pragmatics: We plan to empirically evaluate if link-time compilation can give reasonable "bang for the buck" in the context of a simple prototype compiler.

References

[AB97] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *TACS 97, Sendai, Japan*, 1997.

[AF97] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Prog.*, 3(7), May 1997.

[AFM+95] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 233–246, 1995.

[AK97] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inf. & Comput.*, 139(2):154–233, 15 Dec. 1997.

[AZ99] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, ed., *Proc. Int'l Conf. on Principles and Practice of Declarative Programming*, LNCS, Paris, France, 29 Sept. – 1 Oct. 1999. Springer-Verlag.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[Car97] L. Cardelli. Program fragments, linking, and modularization. In POPL '97 [POPL97].

[CHP97] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. ACM SIGPLAN '97 Conf. Prog. Lang. Design & Impl.*, 1997.

[DEW99] S. Dossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus – towards a model of separate compilation, linking, and binary compatibility. In *Proc. 14th Ann. IEEE Symp. Logic in Computer Sci.*, July 1999.

[DS96] D. Duggan and C. Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*, pp. 262–273, 1996.

[Fer95] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proc. ACM SIGPLAN '95 Conf. Prog. Lang. Design & Impl.*, pp. 103–115, 1995.

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, 1998.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comp. Sc.*, 102:235–271, 1992.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL '94 [POPL94], pp. 123–137.

[Jim96] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In POPL '94 [POPL94], pp. 109–122.

[Ler96] X. Leroy. A modular module system. Tech. Rep. 2866, INRIA, Apr. 1996.

[MT00] E. Machkasova and F. Turbak. A calculus for link-time compilation. Technical report, Comp. Sci. Dept., Boston Univ., 2000.

[PC97] M. P. Plezbert and R. K. Cytron. Is "just in time" = "better late than never"? In POPL '97 [POPL97], pp. 120–131.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theor. Comp. Sc.*, 1:125–159, 1975.

[POPL94] *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, 1994.

[POPL97] *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[SA93] Z. Shao and A. Appel. Smartest recompilation. In *Conf. Rec. 20th Ann. ACM Symp. Princ. of Prog. Langs.*, 1993.

[WV99] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). Full paper with three appendices for proofs, Aug. 1999.