JFP **13** (3): 647–676, May 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796802004616 Printed in the United Kingdom

Inlining as staged computation

STEFAN MONNIER and ZHONG SHAO
Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA
(*e-mail:* {monnier,shao}@cs.yale.edu)

Abstract

Inlining and specialization appear in various forms throughout the implementation of modern programming languages. From mere compiler optimizations to sophisticated techniques in partial evaluation, they are omnipresent, yet each application is treated differently. This paper is an attempt at uncovering the relations between inlining (as done in production compilers) and staged computation (as done in partial evaluators) in the hope of bringing together the research advances in both fields. Using a two-level lambda calculus as the intermediate language, we show how to model inlining as a staged computation while avoiding unnecessary code duplication. The new framework allows us to define inlining annotations formally and to reason about their interactions with module code. In fact, we present a cross-module inlining algorithm that inlines all functions marked inlinable, even in the presence of ML-style parameterized modules.

1 Introduction

Clear and maintainable code requires modularity and abstraction to enforce well-designed interfaces between software components. The module language of Standard ML (Milner *et al*., 1997) provides powerful tools for such high-level code structuring. But these constructs often incur a considerable performance penalty which forces the programmer to break abstraction boundaries or to think twice before using advanced features like parameterized modules (e.g. ML functors).

Efficient implementation of these high-level language constructs often rely crucially on function inlining. Inlining algorithms have been used for many years, but their 'best-effort' behavior prevents us from knowing or making sure that a function will always be inlined (at least, wherever possible given the compilation model). For example, SML/NJ (Appel & MacQueen, 1991) has several *ad hoc* tricks sprinkled in the code to expand primitive operations. These tricks tend to muddy up the abstraction boundaries so it would be nice if they could be replaced by a general-purpose inlining algorithm.

But would the inliner perform as good a job inlining those primitive operations as with the *ad hoc* approaches? For simple cases, it is straightforward to ensure that primitive operations are always inlined, but when higher-order functions or even higher-order modules (such as SML/NJ functors or Java generics) come into play, coupled with separate compilation, the question becomes more challenging. In the course of implementing an extension of Blume and Appel's cross-module

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

inlining algorithm (1997), we tried to understand the relationship between inlining opportunities and separate compilation. We felt a need to formalize our solution to better understand its behavior.

This paper is the result of our efforts to formalize our inlining algorithm. More specifically, we borrow from the partial-evaluation community (Jones *et al*., 1993) to model inlining as a staged computation. By using a two-level λ-calculus (Moggi, 1997) as our intermediate language, we can assign each function (in our program) with a binding-time annotation of *static* or *dynamic*: a static function call is executed at compile time thus is always inlined, while a dynamic call is executed at run time thus is not inlined. The inlining optimization is then equivalent to the standard off-line partial evaluation: first use the binding-time analysis to locate all the inlining candidates, then run the specialization phase to do the actual inlining. The binding-time attributes can also be exported to the source level (or the compiler front-end) to serve as inlining annotations and to allow programmers (or the compiler writer) to control various inlining decisions manually.

Apparently, all partial evaluators support some form of β-reductions as part of the specialization, however, these techniques do not immediately apply to the inlining optimization. Because of the different application domains, partial evaluators are generally much more aggressive than compiler optimizers. Even the binding time annotations can pose problems at the source level because they can clutter the module interface and interact badly with ML functors; for example, we would have to add abstraction over binding-time (commonly called 'binding-time polymorphism') in the type if we want to apply a functor to modules with different binding time (but with same signature otherwise).

The main objective of this paper is to hammer out these details and to see what it would take to launch various partial-evaluation techniques into real compilers. Our paper builds upon previous work on cross-module inlining (Blume & Appel, 1997; Shao, 1998; Leroy, 1995) and two-level λ-calculus (Moggi, 1997; Nielson & Nielson, 1992; Davies & Pfenning, 1996; Taha & Sheard, 1997) but makes the following new contributions:

- As far as we know, our work is the first comprehensive study on how to model inlining as staged computation. The formalism from the staging calculus allows us to explicitly reason about and manipulate the set of inlinable functions. Doing such reasoning is much harder with a traditional inlining algorithm, especially in the presence of ML-style parameterized modules.
- By careful engineering of binding-time coercions, and combined with proper staging and splitting, we show how to model inlining as staged computation without introducing unwanted code duplication (see section 5).
- Adding inlining annotations to a surface language allows a programmer to mark a function as inlinable explicitly. Inlining annotations, however, could pollute the module interface with additional binding-time specifications. This makes the underlying module language more complex. We show how to support inlining annotations while still avoiding such pollution. In fact, our scheme is fully compatible with ML-style modules and requires no change to its module language.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

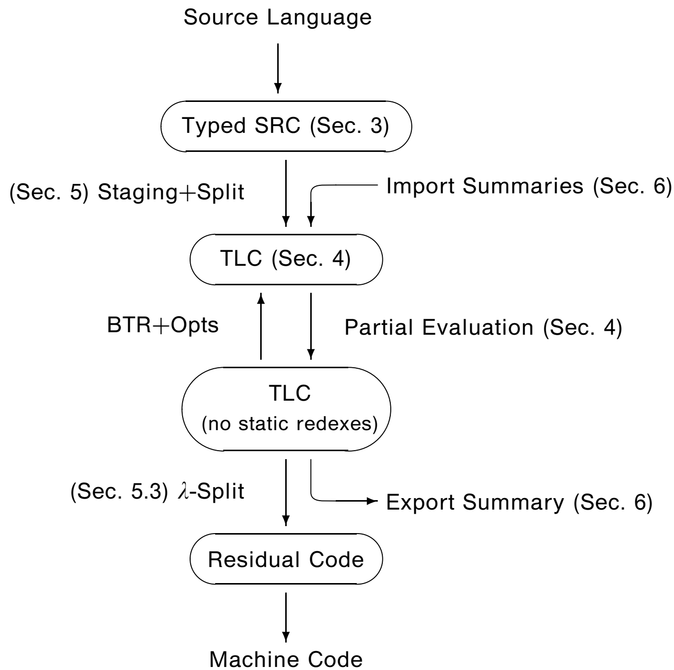


Fig. 1. Structure of the compiler.

- Using a two-level λ-calculus, we show how inlining annotations are compiled into the internal binding-time annotations and how they interact with the module code. This allows us to propagate inlining information across arbitrarily functorized code, even in the presence of separate compilation.
- We extend binding-time coercions to work with parametric polymorphism.

The rest of this paper is organized as follows: Section 2 gives an overview of a compiler that supports cross-module inlining and shows how inlining annotations (at the source level) and two-level λ-calculus (as intermediate language) fit into the picture. Section 3 formally defines our source language SRC which supports inlining annotations and (indirectly) ML-style modules, and section 4 formally defines our target language TLC which is a two-level λ-calculus supporting staged computation. Section 5 presents our detailed algorithm for compiling SRC programs into TLC; the algorithm involves staging, splitting, and careful insertion of binding-time coercions. Section 6 shows how to handle top-level issues for inlining across multiple compilation units. Section 7 then presents several extensions over the basic algorithm. Finally, sections 8 and 9 describe related work and then conclude.

2 The Big Picture

To model inlining as staged computation, we first give an overview of a compiler that supports cross-module inlining. We use our FLINT compiler (Shao, 1997b;

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

Shao *et al.*, 1998) as an example. Figure 1 shows various stages of compilation used in the compiler. The source code is first turned into a strongly typed intermediate language based on a predicative System-F calculus (we name it SRC and present its details in section 3). The SRC calculus contains a module language and a core language. Each core-language function is annotated with inlining hints to indicate whether the function should be inlined or not. Those hints could be provided by the user or by the earlier phases of a compiler (using some inlining heuristics).

The inlining hints are then turned into staging annotations, mapping inlinable functions to static functions (functions executed at compile-time) and the rest to dynamic code (executed at run-time), by translating the code into a two-level intermediate language extended with polymorphism (we name it TLC and present its details in section 4).

To minimize the performance cost of the module code, we want to mark it as static so as to expose as many inlining opportunities as possible. But this would imply that each functor application (SML's equivalent to template instantiation) would create a duplicate copy of the full functor body. This approach, while sometimes acceptable, can lead to excessive code growth and compilation times for heavily functorized code, as any programmer who has worked with C++ templates knows.

We use a variant of the λ-splitting technique (Blume & Appel, 1997) to split each module function into a static part and a dynamic part. This splitting is done carefully to ensure that it does not obfuscate any inlining opportunity. Splitting is done together with staging in the main translation algorithm (see section 5.4). The resulting code is completed by incorporating a copy of the *summaries* from all the import modules (see section 6); a *summary* contains the code that should be inlined across compilation-units, similarly to OCaml's *approximations* (Leroy, 1995).

The static part of the code is then reduced by a straightforward partial evaluation returning the same code but with no remaining static redexes. This code then goes through a Binding-Time Refinement (BTR) or other optimization phases which could introduce new static code requiring a new pass through the partial evaluator.

Once these optimization steps are finished, we reuse the λ-splitting algorithm to split the compilation-unit itself into a *summary* containing all the remaining static code (i.e. inlinable code for future cross-compilation-unit inlining) and a fully dynamic residual code (encompassing the bulk of the program) which is then passed to the code generator.

Inlining across compilation units increases the coupling between those units. If a unit is modified, all units that import it will now need to be recompiled, even if the modification was only internal and did not change the interface. This is automatically handled in our case by a compilation manager (Blume, 1995).

3 The source calculus SRC

This section formally defines our source language SRC which is a variant of the polymorphic lambda calculus System-F (Girard, 1972; Reynolds, 1974). SRC differs from System-F in that it has inlining annotations on the core functions and it has a stratified structure with a polymorphic module language layered on top of a monomorphic core language. Also the module language uses A-normal

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$
\begin{array}{lll}
(\mathit{ctypes}) & \tau & ::= \mathsf{int} \mid t \mid \tau_1 \rightarrow \tau_2 \\
(\mathit{mtypes}) & \sigma & ::= \mathsf{V}(\tau) \mid \langle\, \sigma_1, \ldots, \sigma_n \rangle \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t.\sigma \\
(\mathit{inline}) & a & ::= \quad \mid \mathtt{i} \\
(\mathit{cterms}) & c & ::= n \mid z \mid \pi_{\mathsf{v}} x \mid \lambda^a z : \tau.c \mid c_1 c_2 \\
(\mathit{mterms}) & m & ::= x \mid \iota_{\mathsf{v}}(c) \mid \langle\, x_1, \ldots, x_n \rangle \mid \pi_i x \mid \lambda x : \sigma.m \mid @ x_1 x_2 \mid \Lambda t.m \mid x[\tau] \\
 & & \quad \mid \mathsf{let}\ x = m_1\ \mathsf{in}\ m_2
\end{array}
$$

Fig. 2. Syntax for the source calculus SRC.

form (Flanagan *et al.*, 1993) which means that all intermediate values need to be named via let-binding, thus making all sharing between expressions explicit.

The syntax of SRC is given in figure 2. Here, an SRC program is just a module term (m). Each module term can be either a variable (x), a structure ($\iota_{\mathsf{v}}(c)$) consisting of a single core term (c), a compound module consisting of a collection of other modules ($\langle\, x_1, \ldots, x_n \rangle$), an i-th component from another module ($\pi_i x$), a parameterized module (over other modules: $\lambda x : \sigma.m$ or over types: $\Lambda t.m$), a module application (over other modules: $@ x_1 x_2$ or over types: $x[\tau]$), or a let declaration.

Because the module language can already express polymorphic functions, we intentionally restrict the core language to be a simply typed lambda calculus. A core term can be either an integer constant (n), a variable (z), a value field of a module ($\pi_{\mathsf{v}} x$), a function definition ($\lambda^a z : \tau.c$) with inlining annotation (a), or a function application ($c_1 c_2$).

A module type can either be a singleton-value type ($\mathsf{V}(\tau)$ which refers to a module consisting of a core term of type τ), a compound module type ($\langle\, \sigma_1, \ldots, \sigma_n \rangle$ with n sub-modules, each with type σ_i for $i = 1, \ldots, n$), or a parameterized module (over other modules: $\sigma_1 \rightarrow \sigma_2$ or over types: $\forall t.\sigma$). A core type can be either the integer type (int), a type variable, or a function type ($\tau_1 \rightarrow \tau_2$). The singleton-value type $\mathsf{V}(\tau)$ is used to distinguish between cases like $\mathsf{V}(\mathsf{int} \rightarrow \mathsf{int})$ and $\mathsf{V}(\mathsf{int}) \rightarrow \mathsf{V}(\mathsf{int})$.

The SRC language was chosen to be expressive enough to exhibit the main difficulties that an optimizer based on staged computation might encounter. The language is split between the module and the core languages because the inliner needs to use two different compilation strategies. Of course, we could merge the two languages and annotate the terms to indicate whether or not to treat them like module code. Recent work (Shao, 1998; Shao, 1999; Harper *et al.*, 1990) has shown that Standard ML can be compiled into an SRC-like typed intermediate language.

Figure 3 gives the static semantics for SRC. The environment Δ is the list of bound type variables; the type environment Γ maps both core and module variables to their respective types. Both the type- and the term-formation rules are rather straight-forward. The language is predicative in that the module language supports polymorphism but type variables can only be instantiated to core types. SRC uses a call-by-value semantics (omitted since it is straightforward); it is easy to show that the typing system for SRC is sound with respect to the corresponding dynamic semantics.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$\text{kind environment}\, \Delta ::= \cdot \mid \Delta, t$$
$$\text{type environment}\, \Gamma ::= \cdot \mid \Gamma, z : \tau \mid \Gamma, x : \sigma$$

$\boxed{\Delta \vdash \tau \text{ and } \Delta \vdash \sigma \text{ and } \Delta \vdash \Gamma}$

$$\frac{}{\Delta \vdash \mathsf{int}} \qquad \frac{t \in \Delta}{\Delta \vdash t} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \mathsf{V}(\tau)} \qquad \frac{\Delta \vdash \sigma_i \quad (1 \leqslant i \leqslant n)}{\Delta \vdash \langle \sigma_1, \ldots, \sigma_n \rangle}$$

$$\frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2} \qquad \frac{\Delta, t \vdash \sigma}{\Delta \vdash \forall t.\sigma} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \tau}{\Delta \vdash \Gamma, z : \tau} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma}{\Delta \vdash \Gamma, x : \sigma}$$

$\boxed{\Delta; \Gamma \vdash m : \sigma \text{ and } \Delta; \Gamma \vdash c : \tau}$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash z : \Gamma(z)} \qquad \frac{\Delta; \Gamma \vdash x : \mathsf{V}(\tau)}{\Delta; \Gamma \vdash \pi_{\mathsf{v}} x : \tau}$$

$$\frac{\Delta; \Gamma, z : \tau_1 \vdash c : \tau_2}{\Delta; \Gamma \vdash \lambda^a z : \tau_1.c : \tau_1 \to \tau_2} \qquad \frac{\Delta; \Gamma \vdash c_1 : \tau_2 \to \tau_1 \quad \Delta; \Gamma \vdash c_2 : \tau_2}{\Delta; \Gamma \vdash c_1 c_2 : \tau_1}$$

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta; \Gamma \vdash c : \tau}{\Delta; \Gamma \vdash \iota_{\mathsf{v}}(c) : \mathsf{V}(\tau)} \qquad \frac{\Delta; \Gamma \vdash x_i : \sigma_i \quad (1 \leqslant i \leqslant n)}{\Delta; \Gamma \vdash \langle x_1, \ldots, x_n \rangle : \langle \sigma_1, \ldots, \sigma_n \rangle}$$

$$\frac{\Delta; \Gamma \vdash x : \langle \sigma_1, \ldots, \sigma_n \rangle \quad 1 \leqslant i \leqslant n}{\Delta; \Gamma \vdash \pi_i x : \sigma_i} \qquad \frac{\Delta; \Gamma, x : \sigma_1 \vdash m : \sigma_2}{\Delta; \Gamma \vdash \lambda x : \sigma_1.m : \sigma_1 \to \sigma_2}$$

$$\frac{\Delta; \Gamma \vdash x_1 : \sigma_2 \to \sigma_1 \quad \Delta; \Gamma \vdash x_2 : \sigma_2}{\Delta; \Gamma \vdash @x_1\, x_2 : \sigma_1} \qquad \frac{\Delta, t; \Gamma \vdash m : \sigma}{\Delta; \Gamma \vdash \Lambda t.m : \forall t.\sigma}$$

$$\frac{\Delta; \Gamma \vdash x : \forall t.\sigma}{\Delta; \Gamma \vdash x[\tau] : \{\tau/t\}\sigma} \qquad \frac{\Delta; \Gamma \vdash m_1 : \sigma_1 \quad \Delta; \Gamma, x : \sigma_1 \vdash m_2 : \sigma_2}{\Delta; \Gamma \vdash \mathsf{let}\ x = m_1\ \mathsf{in}\ m_2 : \sigma_2}$$

Fig. 3. Static semantics for SRC.

The most interesting feature of SRC is the inlining annotation a. The annotation `i` means that the underlying lambda expression should be inlined while the empty annotation means it should not. Notice that we do not track inlining annotations in the types; a core function $\lambda^a z : \tau.c$ is still assigned with the same type whether the annotation a is `i` or empty.

This design choice is deliberate. We believe inlining annotations should be made as non-intrusive as possible. Tracking them in the types would significantly complicate the module language; for example, we would have to add binding-time polymorphism in the type if we want to apply a functor to modules with different inlining annotations (but with the same signature otherwise).

When compiling SML to an SRC-like language, the SML module language maps to the SRC module language as expected, but polymorphic core SML functions also map to module-level type abstractions (together with a core-level function) in SRC.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$
\begin{array}{llll}
(\textit{kind}) & b,k & ::= & \mathsf{s} \mid \mathsf{d} \\
(\textit{type}) & \sigma & ::= & \mathsf{int} \mid t \mid \langle \sigma_1,\ldots,\sigma_n\rangle^b \mid \sigma_1 \xrightarrow{b} \sigma_2 \mid \forall^b t : k.\sigma \\
(\textit{term}) & e & ::= & v \mid \pi_i^b v \mid @^b v_1\, v_2 \mid v[\sigma]^b \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
(\textit{value}) & v & ::= & n \mid x \mid \langle v_1,\ldots,v_n\rangle^b \mid \lambda^b x : \sigma.e \mid \Lambda^b t : k.e
\end{array}
$$

Fig. 4. Syntax for the target calculus TLC.

This does not introduce any problem, however; since polymorphic recursion is not available, type instantiations can be done statically, or hoisted to the top-level (Saha & Shao, 1998).

4 The target calculus TLC

This section formally defines our target language TLC. As a typed intermediate language, TLC is is essentially a hybrid of System-F in A-normal form (Flanagan *et al*., 1993) and the two-level lambda calculus $\lambda 2sd$ by Moggi (1997).

The syntax of TLC is given in figure 4. A TLC term (e) can be either a value (v), a record selection, a function application, a type application, or a let expression. A TLC value (v) is either an integer constant, a variable, an n-tuple, a function, or a type function. A TLC type is either the integer type, a type variable, a record type, a function type, or a polymorphic type. Many of these are annotated with a binding-time annotation (called 'kind') that can either be s for static code (evaluated at compile-time) or d for dynamic code (evaluated at run-time).

Compared to SRC, TLC replaces inlining hints on core functions with staging annotations on tuples, functions and type-abstractions and merges the module and the core languages since the distinction between the two is only needed to direct the translation from SRC. Notice also how $\forall$ types have two binding-time annotations, one for the type abstraction itself and another that constrains the possible types it can be instantiated to.

To simplify the presentation, we force the ground types (i.e. int) to be considered as dynamic. This is justified by the fact that we are only interested in function-level reductions. We may lift this restriction if we want to model constant propagation.

In the rest of this paper, we will also use the following syntactic sugar:

$$
\begin{array}{rcl}
\pi_i^b e & \equiv & \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i^b x \\
e[\sigma]^b & \equiv & \mathsf{let}\ x = e\ \mathsf{in}\ x[\sigma]^b \\
@^b e_1\, e_2 & \equiv & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ @^b x_1\, x_2 \\
\langle e_1,\ldots,e_n\rangle^b & \equiv & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ \langle x_1,\ldots,x_n\rangle^b \\
\lambda^b \langle x_1,\ldots,x_n\rangle^{b'} : \sigma.e & \equiv & \lambda^b x : \sigma.\mathsf{let}\ x_1 = \pi_1^{b'} x\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = \pi_n^{b'} x\ \mathsf{in}\ e
\end{array}
$$

Essentially, we will put an e term where only v is allowed, leaving the let transformation implicit and we will use a pattern-matching variant of let; we will also assume that alpha-renaming is used so variables are never shadowed.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$\text{kind environment}\,\Delta ::= \cdot \mid \Delta, t{:}k$$
$$\text{type environment}\,\Gamma ::= \cdot \mid \Gamma, x{:}\sigma$$

$\boxed{b_1 \leqslant b_2}$

$$\mathsf{d} \leqslant \mathsf{d} \qquad \mathsf{d} \leqslant \mathsf{s} \qquad \mathsf{s} \leqslant \mathsf{s}$$

$\boxed{\Delta \vdash \sigma : k \text{ and } \Delta \vdash \Gamma}$

$$\Delta \vdash \mathsf{int} : \mathsf{d} \qquad \Delta \vdash t : \Delta(t) \qquad \frac{\Delta \vdash \sigma : b_1 \quad b_1 \leqslant b_2}{\Delta \vdash \sigma : b_2} \qquad \frac{\Delta \vdash \sigma_i : b \;\; (1 \leqslant i \leqslant n)}{\Delta \vdash \langle \sigma_1, \ldots, \sigma_n \rangle^b : b}$$

$$\frac{\Delta \vdash \sigma_1 : b \quad \Delta \vdash \sigma_2 : b}{\Delta \vdash \sigma_1 \xrightarrow{b} \sigma_2 : b} \qquad \frac{\Delta, t{:}k \vdash \sigma : b \quad k \leqslant b}{\Delta \vdash \forall^b t{:}k.\sigma : b} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma : \mathsf{s}}{\Delta \vdash \Gamma, x{:}\sigma}$$

$\boxed{\Delta;\Gamma \vdash e : \sigma \text{ and } \Delta;\Gamma \vdash v : \sigma}$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta;\Gamma \vdash v_1 : \sigma_2 \xrightarrow{b} \sigma_1 \quad \Delta;\Gamma \vdash v_2 : \sigma_2}{\Delta;\Gamma \vdash @^b v_1\, v_2 : \sigma_1}$$

$$\frac{\Delta;\Gamma \vdash v_i : \sigma_i \;\; (1 \leqslant i \leqslant n) \quad \Delta \vdash \langle \sigma_1, \ldots, \sigma_n \rangle^b : b}{\Delta;\Gamma \vdash \langle v_1, \ldots, v_n \rangle^b : \langle \sigma_1, \ldots, \sigma_n \rangle^b} \qquad \frac{\Delta;\Gamma \vdash v : \langle \sigma_1, \ldots, \sigma_n \rangle^b \quad 1 \leqslant i \leqslant n}{\Delta;\Gamma \vdash \pi_i^b v : \sigma_i}$$

$$\frac{\Delta;\Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \quad \Delta \vdash \sigma_1 \xrightarrow{b} \sigma_2 : b}{\Delta;\Gamma \vdash \lambda^b x{:}\sigma_1.e : \sigma_1 \xrightarrow{b} \sigma_2} \qquad \frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \quad \Delta;\Gamma, x{:}\sigma_1 \vdash e_2 : \sigma_2}{\Delta;\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma_2}$$

$$\frac{\Delta, t{:}k;\Gamma \vdash e : \sigma \quad \Delta \vdash \forall^b t{:}k.\sigma : b}{\Delta;\Gamma \vdash \Lambda^b t{:}k.e : \forall^b t{:}k.\sigma} \qquad \frac{\Delta;\Gamma \vdash v : \forall^b t{:}k.\sigma_2 \quad \Delta \vdash \sigma_1 : k}{\Delta;\Gamma \vdash v[\sigma_1]^b : \{\sigma_1/t\}\sigma_2}$$

Fig. 5. Static semantics for TLC.

Figure 5 gives the typing rules for TLC. In addition to the usual type safety, these rules also ensure binding-time correctness. Here the kind environment Δ maps type variables to their binding time; the type environment Γ maps variables to their types. To enforce the usual restriction that no dynamic entity can contain or manipulate a static value, types are classified as being either of dynamic kind or static kind, with a subkind relationship between the two: a type σ of dynamic kind can also be considered to have static kind but not vice versa.

Figures 6–9 give the dynamic semantics for TLC as a set of primitive reductions and single-step evaluation relations that determine where those reductions can be applied. Figure 6 defines the primitive static reduction $e \rightsquigarrow_{\mathsf{s}} e'$. Figure 7 defines the single-step partial evaluation $e \mapsto_{\mathsf{s}} e'$ together with the corresponding $v \mapsto_{\mathsf{s}}^{\mathsf{v}} v'$ used for values. Note how partial evaluation in this language amounts to reducing all the static redexes of a term. Figures 8 and 9 show the corresponding reductions $\rightsquigarrow$ and $\mapsto$ of a standard call-by-value evaluator. In contrast to the partial evaluation case, those reductions apply to both static and dynamic redexes but only to the outermost ones.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$
\begin{array}{llll}
(\beta_\lambda) & @^{\mathsf{s}}(\lambda^{\mathsf{s}}x:\sigma.e)\,(v) & \rightsquigarrow_{\mathsf{s}} \{v/x\}e & \\
(\beta_\Lambda) & (\Lambda^{\mathsf{s}}t:k.e)[\sigma]^{\mathsf{s}} & \rightsquigarrow_{\mathsf{s}} \{\sigma/t\}e & \\
(\pi) & \pi_i^{\mathsf{s}}\langle v_1,\ldots,v_n\rangle^{\mathsf{s}} & \rightsquigarrow_{\mathsf{s}} v_i & \text{if } 1\leqslant i\leqslant n \\
(let) & \mathsf{let}\ x=v\ \mathsf{in}\ e & \rightsquigarrow_{\mathsf{s}} \{v/x\}e & \\
(asc) & \mathsf{let}\ x_2=(\mathsf{let}\ x_1=e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ e_3 & & \\
 & & \rightsquigarrow_{\mathsf{s}}\ \mathsf{let}\ x_1=e_1\ \mathsf{in}\ \mathsf{let}\ x_2=e_2\ \mathsf{in}\ e_3 &
\end{array}
$$

Fig. 6. Primitive static reduction for TLC

$$
\begin{array}{lll}
e \rightsquigarrow_{\mathsf{s}} e' & \Rightarrow & e \mapsto_{\mathsf{s}} e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \mathsf{let}\ x=e\ \mathsf{in}\ e_2 \mapsto_{\mathsf{s}} \mathsf{let}\ x=e'\ \mathsf{in}\ e_2 \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \mathsf{let}\ x=e_2\ \mathsf{in}\ e \mapsto_{\mathsf{s}} \mathsf{let}\ x=e_2\ \mathsf{in}\ e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \lambda^b x:\sigma.e \mapsto_{\mathsf{s}}^{\mathsf{v}} \lambda^b x:\sigma.e' \\
e \mapsto_{\mathsf{s}} e' & \Rightarrow & \Lambda^b t:k.e \mapsto_{\mathsf{s}}^{\mathsf{v}} \lambda^b t:k.e' \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & \langle v_1,\ldots,v,\ldots,v_n\rangle^b \mapsto_{\mathsf{s}}^{\mathsf{v}} \langle v_1,\ldots,v',\ldots,v_n\rangle^b \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & @^b v\, v_2 \mapsto_{\mathsf{s}} @^b v'\, v_2 \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & @^b v_2\, v \mapsto_{\mathsf{s}} @^b v_2\, v' \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & v[\sigma]^b \mapsto_{\mathsf{s}} v'[\sigma]^b \\
v \mapsto_{\mathsf{s}}^{\mathsf{v}} v' & \Rightarrow & \pi_i^b v \mapsto_{\mathsf{s}} \pi_i^b v'
\end{array}
$$

Fig. 7. Single-step partial evaluation for TLC

$$
\begin{array}{llll}
(\rightsquigarrow_{\mathsf{s}}) & e & \rightsquigarrow e' & \text{if } e \rightsquigarrow_{\mathsf{s}} e' \\
(\beta_\lambda) & @^{\mathsf{d}}(\lambda^{\mathsf{d}}x:\tau.e)\,(v) & \rightsquigarrow \{v/x\}e & \\
(\beta_\Lambda) & (\Lambda^{\mathsf{d}}t:k.e)[\tau]^{\mathsf{d}} & \rightsquigarrow \{\tau/t\}e & \\
(\pi) & \pi_i^{\mathsf{d}}\langle v_1,\ldots,v_n\rangle^{\mathsf{d}} & \rightsquigarrow v_i & \text{if } 1\leqslant i\leqslant n
\end{array}
$$

Fig. 8. Primitive reduction relation for TLC

$$
\begin{array}{lll}
e \rightsquigarrow e' & \Rightarrow & e \mapsto e' \\
e \mapsto e' & \Rightarrow & \mathsf{let}\ x=e\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ x=e'\ \mathsf{in}\ e_2
\end{array}
$$

Fig. 9. Single-step call-by-value standard evaluation for TLC.

TLC is a variant of Moggi's computational lambda calculus λ_c (1988) restricted to A-normal form; in fact, the primitive reduction relations in figures 6 and 8 are same as that for λ_c (except that we added type applications and removed η-reductions). We can easily show that the type system for TLC is sound and the static reduction $\mapsto_{\mathsf{s}}$ is strongly normalizing and confluent. We can thus define a partial evaluation function $\mathscr{P}_{\mathsf{e}}(e)$ that returns the static normal form of e. Similarly it is easy to show that $\mapsto$ is confluent, so we can also define a partial function $\mathscr{R}_{\mathsf{e}}(e)$ which does the standard evaluation of e:

$$
\begin{aligned}
\mathscr{P}_{\mathsf{e}}(e) &= e' \text{suchthat} e \mapsto_{\mathsf{s}}^{*} e' \text{andthereisno} e\text{"forwhich} e' \mapsto_{\mathsf{s}} e'' \\
\mathscr{R}_{\mathsf{e}}(e) &= e' \text{suchthat} e \mapsto^{*} e' \text{andthereisno} e\text{"forwhich} e' \mapsto e''
\end{aligned}
$$

where $\mapsto^*$ and $\mapsto_{\mathsf{s}}^*$ are the reflexive transitive closures of $\mapsto$ and $\mapsto_{\mathsf{s}}$. TLC satisfies the following important residualization property:

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

Theorem 4.1 (*Residualization*)
If $\Delta;\Gamma \vdash e : \sigma$ and $\Delta \vdash \sigma : \mathsf{d}$ and $\forall x \in \mathsf{fv}(e).\ \Delta \vdash \Gamma(x) : \mathsf{d}$, then $\mathscr{P}_{\mathsf{e}}(e)$ is free of any static subterms.

In other words, given an expression e with dynamic type σ, partially evaluating e will inline all of its inlinable functions and result in an expression free of static subterms.

Next we show why inlining does not affect the semantics of the program. We first introduce a notion of semantic equivalence on well-typed TLC values:

Definition 4.1 (*Equivalence*)
If $\cdot;\cdot \vdash v : \sigma$ and $\cdot;\cdot \vdash v' : \sigma$, we say that $v \simeq v'$ if one of the following holds:

(int) $v \equiv v'$.
($\times$) $v = \langle v_1, \ldots, v_n \rangle^b$ and $v' = \langle v'_1, \ldots, v'_n \rangle^b$ and $\forall i \in [1..n].v_i \simeq v'_i$.
($\rightarrow$) $\sigma = \sigma_1 \xrightarrow{b} \sigma_2$ and for any value w of type σ_1 then $\mathscr{R}_{\mathsf{e}}(@^b v\, w) \simeq \mathscr{R}_{\mathsf{e}}(@^b v'\, w)$.
($\forall$) $\sigma = \forall^b t : k.\sigma_1$ and for any well-formed type $\sigma_2 : k$ then $\mathscr{R}_{\mathsf{e}}(v[\sigma_2]^b) \simeq \mathscr{R}_{\mathsf{e}}(v'[\sigma_2]^b)$.

The correctness theorem can then be proved by induction over the reduction steps of $\mathscr{P}_{\mathsf{e}}(e)$.

Theorem 4.2 (*Correctness*)
If $\cdot;\cdot \vdash e : \sigma$ and $\cdot \vdash \sigma : \mathsf{d}$ then $\mathscr{R}_{\mathsf{e}}(\mathscr{P}_{\mathsf{e}}(e)) \simeq \mathscr{R}_{\mathsf{e}}(e)$.

5 Translation from SRC to TLC

The translation from SRC to TLC involves both staging and splitting, executed in an interleaved manner. Staging translates inlining annotations in the core language into binding-time annotations. It also calls the splitting algorithm to divide each module term into a static part and a dynamic part. The static part is used to propagate inlining information and implement cross-module inlining. In the rest of this section, we first give a quick overview of our approach; we then show how to stage core terms and split module terms; finally, we give the main translation algorithm that links all the parts together.

5.1 A quick overview

The translation from SRC to TLC mostly consists of adding staging annotations. This is usually known as binding-time analysis and has been extensively studied in the partial evaluation community.

One desirable goal is to make sure that binding-time annotations do not hide opportunities for static evaluation. For example, let us take the inlinable compose function o defined as follows:

$$o = \lambda^{\mathsf{i}} f : \tau_1 \rightarrow \tau_2.\lambda^{\mathsf{i}} g : \tau_2 \rightarrow \tau_3.\lambda x : \tau_1.g(f\,x)$$

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

When translating it, we probably do not want to assign it the following type:

$$o : (\tau_1 \xrightarrow{\mathsf{d}} \tau_2) \xrightarrow{\mathsf{s}} (\tau_2 \xrightarrow{\mathsf{d}} \tau_3) \xrightarrow{\mathsf{s}} (\tau_1 \xrightarrow{\mathsf{d}} \tau_3)$$

(i.e. a static function that composes two dynamic functions) since it would force us to make sure that all the functions passed to it are not inlinable, which mostly defeats the purpose of inlining it in the first place. Now clearly, if we mark it as:

$$o : (\tau_1 \xrightarrow{\mathsf{s}} \tau_2) \xrightarrow{\mathsf{s}} (\tau_2 \xrightarrow{\mathsf{s}} \tau_3) \xrightarrow{\mathsf{s}} (\tau_1 \xrightarrow{\mathsf{d}} \tau_3)$$

that will make it impossible to call it with a non-inlinable function. We could work around this problem by using polymorphism at the binding-time level (Henglein & Mossin, 1994; Glynn *et al.*, 2001), but we decided to keep our calculus simple. With monomorphic staging annotations, we have two options: code duplication to provide a poor man's polymorphic binding-time, or coercions in the form of binding-time improvements (Danvy *et al.*, 1996; Danvy, 1996).

A compiler needs to be very careful about duplicating code so we decided to use coercions instead, especially since they provide us with a lot of flexibility. More specifically, we can completely avoid the need for a full-blown binding-time analysis and use a simple one-pass translation instead, by optimistically marking s any place that might need to accommodate a static value and inserting coercions when needed, just like the unboxing coercions (Leroy, 1992; Shao, 1997a). It also allows us to simplify our types: all types are either (completely) dynamic or completely static.

5.2 Staging the core

Staging could be done via any kind of binding-time analysis (Consel, 1993; Birkedal & Welinder, 1995), but this would be too costly for our application, so instead of performing global code analysis to add the annotations, we add them in a single traversal of the code using only local information. In order to maximize the amount of static computation, we make extensive use of binding-time improvements (Danvy *et al.*, 1996).

Binding-time improvements are usually some form of η-redexes that coerce an object between its static and dynamic representations. They improve the binding-time annotations by allowing values to be used statically at one place and dynamically at another and even to make this choice ‘dynamically’ during specialization.

Staging is then simple: based on the inlining annotations, SRC terms can either be translated to completely static or dynamic entities (except for int, which is always dynamic). Because inlining annotations are not typechecked in SRC, the resulting TLC terms may use dynamic subterms in a static context or vice versa. We insert coercions whenever there is such a mismatch.

We define two type-translation functions $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ that turn any SRC type into either its fully static or its fully dynamic TLC equivalent, and two coercion functions $\downarrow^\tau : |\tau|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{d}}$ and $\uparrow^\tau : |\tau|^{\mathsf{d}} \xrightarrow{\mathsf{s}} |\tau|^{\mathsf{s}}$. Those coercions (and corresponding

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

type translations) could simply be:

$$\begin{aligned}
\downarrow^{\mathsf{int}} x &= x \\
\downarrow^{\tau_1 \to \tau_2} x &= \lambda^{\mathsf{d}} x_1 : |\tau_1|^{\mathsf{d}}.\ \downarrow^{\tau_2}(@^{\mathsf{s}} x\, (\uparrow^{\tau_1} x_1)) \\
&\cdots \\
\uparrow^{\mathsf{int}} x &= x \\
\uparrow^{\tau_1 \to \tau_2} x &= \lambda^{\mathsf{s}} x_1 : |\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}} x\, (\downarrow^{\tau_1} x_1)) \\
&\cdots
\end{aligned}
\qquad
\begin{aligned}
|\mathsf{int}|^{\mathsf{d}} &= \mathsf{int} \\
|\tau_1 \to \tau_2|^{\mathsf{d}} &= |\tau_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\tau_2|^{\mathsf{d}} \\
&\cdots \\
|\mathsf{int}|^{\mathsf{s}} &= \mathsf{int} \\
|\tau_1 \to \tau_2|^{\mathsf{s}} &= |\tau_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau_2|^{\mathsf{s}} \\
&\cdots
\end{aligned}$$

But this would run the risk of introducing unexpected code duplication.

Spurious copies A naive coercion of a static function to its dynamic equivalent tends to introduce static redexes which cause the function to be inlined unnecessarily at the place where it escapes. Consider the following piece of SRC code:

$$\begin{aligned}
\mathsf{let}\ id &= \lambda^{\mathsf{i}} x : \mathsf{int}.x \\
big &= \lambda f : \mathsf{int} \to \mathsf{int}.\ ...big\ body... \\
\mathsf{in}\ &\langle\, id, @big\ id \rangle
\end{aligned}$$

A simple-minded staging scheme would turn it into:

$$\begin{aligned}
\mathsf{let}\ id &= \lambda^{\mathsf{s}} x : \mathsf{int}.x \\
big &= \lambda^{\mathsf{d}} f : \mathsf{int} \xrightarrow{\mathsf{d}} \mathsf{int}.\ ...big\ body... \\
\mathsf{in}\ &\langle\ \downarrow^{\mathsf{int}\to\mathsf{int}} id, @^{\mathsf{d}} big\ (\downarrow^{\mathsf{int}\to\mathsf{int}} id) \rangle^{\mathsf{d}}
\end{aligned}$$

where the coercions get expanded to:

$$\begin{aligned}
\mathsf{let}\ id &= \lambda^{\mathsf{s}} x : \mathsf{int}.x \\
big &= \lambda^{\mathsf{d}} f : \mathsf{int} \xrightarrow{\mathsf{d}} \mathsf{int}.\ ...big\ body... \\
\mathsf{in}\ &\langle\, \lambda^{\mathsf{d}} x : \mathsf{int}.@^{\mathsf{s}} id\ x, @^{\mathsf{d}} big\ (\lambda^{\mathsf{d}} x : \mathsf{int}.@^{\mathsf{s}} id\ x) \rangle^{\mathsf{d}}
\end{aligned}$$

Note that the two escaping uses of *id* have been turned now into η-redexes where *id* is called directly. Thus specialization will happily inline two copies of *id* even though no optimization will be enabled since both uses are really escaping. We do not want to introduce such wasteful code duplication.

In other words, we want to ensure that there can be only one non-inlined copy of any function, shared among all its escaping uses. To this end, we must arrange for $\downarrow^{\sigma}$ not to introduce spurious redexes. We could simply introduce a special coerce primitive operation with an ad-hoc treatment in the partial-evaluator, but depending on the semantics chosen (e.g. binding-time erasure) it can end up leaking static code to run-time, introducing unwanted run-time penalties and it does not easily solve the problem of ensuring a unique dynamic copy of a function, even in the presence of cross-module inlining.

So we decided to choose a fancier representation for the static translation of a function, where each static function is now represented as a pair of the real static function and the already-coerced dynamic function. Now $\downarrow^{\sigma_1 \to \sigma_2}$ becomes π_2^{s} and the only real coercion happens once, making it clear that only one instance of the dynamic version will exist. The definition of our type translations $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ and coercions $\downarrow^{\tau}$ and $\uparrow^{\tau}$ for the core calculus is shown in figure 10. The previous example

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$
\begin{array}{rcl}
|\mathsf{int}|^{\mathsf{d}} & = & \mathsf{int} \\
|\tau_1 \to \tau_2|^{\mathsf{d}} & = & |\tau_1|^{\mathsf{d}} \stackrel{\mathsf{d}}{\to} |\tau_2|^{\mathsf{d}}
\end{array}
\qquad
\begin{array}{rcl}
|\mathsf{int}|^{\mathsf{s}} & = & \mathsf{int} \\
|\tau_1 \to \tau_2|^{\mathsf{s}} & = & \langle\, |\tau_1|^{\mathsf{s}} \stackrel{\mathsf{s}}{\to} |\tau_2|^{\mathsf{s}}, |\tau_1 \to \tau_2|^{\mathsf{d}} \rangle^{\mathsf{s}}
\end{array}
$$

$$
\begin{array}{lcl}
\downarrow^{\tau} & : & |\tau|^{\mathsf{s}} \stackrel{\mathsf{s}}{\to} |\tau|^{\mathsf{d}} \\
\downarrow^{\mathsf{int}} x & = & x \\
\downarrow^{\tau_1 \to \tau_2} x & = & \pi_2^{\mathsf{s}} x
\end{array}
\qquad
\begin{array}{lcl}
\uparrow^{\tau} & : & |\tau|^{\mathsf{d}} \stackrel{\mathsf{s}}{\to} |\tau|^{\mathsf{s}} \\
\uparrow^{\mathsf{int}} x & = & x \\
\uparrow^{\tau_1 \to \tau_2} x & = & \langle\, \lambda^{\mathsf{s}} x_1 : |\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}} x\, (\downarrow^{\tau_1} x_1)), x \rangle^{\mathsf{s}}
\end{array}
$$

$$
\begin{array}{rcl}
|\,\Delta;\Gamma;\Sigma\,| & = & \Delta;\Gamma' \\
\text{where}\Gamma' & = & \{x : |\Gamma(x)|^{\mathsf{s}} \mid x \in \mathsf{dom}(\Gamma)\} \cup \{z : |\tau|^{b} \mid \tau = \Gamma(z) \text{and} b = \Sigma(z)\} \\
\text{and}(\textit{binding-time environment})\ \Sigma & ::= & \cdot \mid \Sigma, z : b
\end{array}
$$

Fig. 10. Core type translations $|\cdot|$ and coercions $\downarrow$ and $\uparrow$.

is now staged as follows:

$$
\begin{array}{l}
\mathsf{let}\ id \ = \langle\, \lambda^{\mathsf{s}} x : \mathsf{int}.x, \lambda^{\mathsf{d}} x : \mathsf{int}.x \rangle^{\mathsf{s}} \\
\quad\ \ big = \lambda^{\mathsf{d}} f : \mathsf{int} \stackrel{\mathsf{d}}{\to} \mathsf{int}.\ ...big\ body... \\
\mathsf{in}\ \langle\ \downarrow^{\mathsf{int} \to \mathsf{int}} id, @^{\mathsf{d}} big\ (\downarrow^{\mathsf{int} \to \mathsf{int}} id) \rangle^{\mathsf{d}}
\end{array}
$$

Since the coercion $\downarrow^{\mathsf{int}\to\mathsf{int}}$ is now π_2^{s}, it just selects the dynamic version of id, with no code duplication.

Partial evaluators have long used such paired representation in their specializer for similar reasons (Asai, 1999), although our case is slightly different in that the pairs are explicit in the program being specialized rather than used internally by the specializer.

This pairing approach can also be seen as a poor man's polymorphic binding-time where we only allow the two extreme cases (all dynamic or all static). Minamide & Garrigue (1998) used the same pairing approach when trying to avoiding the problem of accumulative coercion wrappers that appears when unboxing coercions are used to reconcile polymorphism and specialized data representation.

The staging algorithm is shown in Fig. 11. The judgment $\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{b}{\Rightarrow} e$ says that a core SRC term c of type τ (under contexts Δ and Γ) is translated into a TLC term e. The environment Σ maps core variables to the binding-time of the corresponding variable in e. The b on the arrow indicates whether a static or a dynamic term e is expected.

Most rules come in two forms, depending on whether the context expects a dynamic or static term. The dynamic case is trivial (it corresponds to the no-inlining case so we do not need to do anything) while the static case needs to build the static/dynamic pair (in the λ case) or to extract the static half of the pair before applying it (in the @ case).

5.3 Splitting

Module-level functions are typically used differently from core-level functions. They also do not have any inlining annotations thus deserve special treatment during

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$\boxed{\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{b}{\Rightarrow} e}$ such that $|\Delta;\Gamma;\Sigma| \vdash e : |\tau|^b$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma;\Sigma \vdash [\![n : \mathsf{int}]\!] \stackrel{b}{\Rightarrow} n} \qquad \frac{\Delta \vdash \Gamma \quad \Sigma(z) = b}{\Delta;\Gamma;\Sigma \vdash [\![z : \Gamma(\tau)]\!] \stackrel{b}{\Rightarrow} z} \qquad \frac{\Delta;\Gamma \vdash x : \mathsf{V}(\tau)}{\Delta;\Gamma;\Sigma \vdash [\![\pi_{\mathsf{v}} x : \tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} x}$$

$$\frac{\Delta;\Gamma,z:\tau_1;\Sigma,z:\mathsf{s} \vdash [\![c : \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} e}{\begin{array}{l}\Delta;\Gamma;\Sigma \vdash [\![\lambda^{\mathsf{i}} z:\tau_1.c : \tau_1 \to \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} \\ \mathsf{let}\ x_{\mathsf{s}} = \lambda^{\mathsf{s}} z:|\tau_1|^{\mathsf{s}}.e \\ \quad x_{\mathsf{d}} = \lambda^{\mathsf{d}} z:|\tau_1|^{\mathsf{d}}.\downarrow^{\tau_2}(@^{\mathsf{s}} x_{\mathsf{s}}\,(\uparrow^{\tau_1} z)) \\ \mathsf{in}\ \langle x_{\mathsf{s}}, x_{\mathsf{d}}\rangle^{\mathsf{s}}\end{array}} \qquad \frac{\Delta;\Gamma,z:\tau_1;\Sigma,z:\mathsf{d} \vdash [\![c : \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![\lambda z:\tau_1.c : \tau_1 \to \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} \lambda^{\mathsf{d}} z:|\tau_1|^{\mathsf{d}}.e}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash [\![c_1 : \tau_1 \to \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} e_1 \quad \Delta;\Gamma;\Sigma \vdash [\![c_2 : \tau_1]\!] \stackrel{\mathsf{d}}{\Rightarrow} e_2}{\Delta;\Gamma;\Sigma \vdash [\![c_1 c_2 : \tau_2]\!] \stackrel{\mathsf{d}}{\Rightarrow} @^{\mathsf{d}} e_1\, e_2} \qquad \frac{\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{\mathsf{d}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} \uparrow^{\tau} e}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash [\![c_1 : \tau_1 \to \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} e_1 \quad \Delta;\Gamma;\Sigma \vdash [\![c_2 : \tau_1]\!] \stackrel{\mathsf{s}}{\Rightarrow} e_2}{\Delta;\Gamma;\Sigma \vdash [\![c_1 c_2 : \tau_2]\!] \stackrel{\mathsf{s}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} e_1)\, e_2} \qquad \frac{\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} e}{\Delta;\Gamma;\Sigma \vdash [\![c : \tau]\!] \stackrel{\mathsf{d}}{\Rightarrow} \downarrow^{\tau} e}$$

Fig. 11. Core code translation.

the translation. As noted earlier, it is desirable to mark all the module code as static to 'compile it away' or at least, to allow inlining information to flow freely through module boundaries. But that would imply that every single module-level function application gets its own copy of the body, which leads to unnecessary code duplication.

To overcome this difficulty, we use a form of partial inlining inspired from Blume and Appel's λ-splitting (1997) that splits each function into a static and a dynamic part. It rewrites a TLC expression e into a list of let bindings and copies every inlinable (i.e. static) binding from e into e_{i}, and puts the rest into the expression e_{e} (the e subscript stands for 'expansive') in such a way that the two can be combined to get back an expression equivalent to e with $e \simeq \mathsf{let}\ \langle fv \rangle = e_{\mathsf{e}}\ \mathsf{in}\ e_{\mathsf{i}}$ where fv is the list of free variables of e_{i}. Since e_{i} is small, it can be copied wherever e was originally used, while the main part of the code is kept separate in e_{e}.

Basically, e_{i} is just like e but where all the non-inlinable code has been taken out (and the variables that refer to it are thus free), whereas e_{e} is a complete copy of e except that it returns all those values that have been taken out of e_{i}, so it can be used to close over the free variables of e_{i}. Take for example the following expression e where *lookup* is inlinable but *balance* is not (note that the algorithm assumes that the return value of e is completely static):

$$\begin{array}{l} e = \mathsf{let}\ balance = \lambda^{\mathsf{d}}\langle t, x\rangle^{\mathsf{d}} : \langle tree^{\mathsf{d}}, elem^{\mathsf{d}}\rangle^{\mathsf{d}}.\ldots \\ \qquad\qquad lookup\ = \lambda^{\mathsf{s}}\langle t, p\rangle^{\mathsf{s}} : \langle tree^{\mathsf{s}}, elem^{\mathsf{s}} \stackrel{\mathsf{s}}{\to} bool\rangle^{\mathsf{s}}. \\ \qquad\qquad\qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}} find\ \ p)\, t\ \mathsf{in}\ @^{\mathsf{d}} balance\,\langle\ \downarrow^{tree} t, \downarrow^{elem} x\rangle^{\mathsf{d}} \\ \qquad \mathsf{in}\ lookup \end{array}$$

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$\boxed{\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}$ where $\Delta;\Gamma \vdash e \;:\; |\sigma|^{\mathsf{s}}$

$$\frac{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ e_3]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x_2 = \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2\ \mathsf{in}\ e_3]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}\ (\mathrm{sp-asc})$$

$$\frac{\begin{array}{c} x \notin \mathsf{fv}(e_{\mathsf{i}}) \vee \Delta \vdash \sigma_1 \;:\; \mathsf{d} \\ \Delta;\Gamma \vdash e_1 \;:\; \sigma_1 \quad \Delta;\Gamma, x\!:\!\sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}} \end{array}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \stackrel{\sigma}{\Longrightarrow} (\mathsf{let}\ x = e_1\ \mathsf{in}\ E_{\mathsf{e}}) \,;\, e_{\mathsf{i}}}\ (\mathrm{sp-share})$$

$$\frac{\Delta;\Gamma \vdash e_1 \;:\; \sigma_1 \quad \Delta;\Gamma, x\!:\!\sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}{\begin{array}{c}\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \stackrel{\sigma}{\Longrightarrow} \\ (\mathsf{let}\ x = e_1\ \mathsf{in}\ E_{\mathsf{e}}) \;;\; (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_{\mathsf{i}})\end{array}}\ (\mathrm{sp-dup})$$

$$\frac{E_{\mathsf{e}} = (\mathsf{let}\ x_{\mathsf{fv}} = \bullet\ \mathsf{in}\ \langle\, \downarrow^{\sigma} x,\, x_{\mathsf{fv}}\rangle^{\mathsf{d}})}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![x]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, x}\ (\mathrm{sp-var}) \qquad \frac{\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e\ \mathsf{in}\ x]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}{\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}} \,;\, e_{\mathsf{i}}}\ (\mathrm{sp-exp})$$

Fig. 12. The λ-split algorithm.

This expression e will be split into a dynamic e_{e} and a static e_{i} where *balance* has been taken out since it is not inlinable. e_{i} will look like:

$$\begin{array}{l} e_{\mathsf{i}} = \mathsf{let}\ \mathit{lookup} = \lambda^{\mathsf{s}}\langle\, t, p\rangle^{\mathsf{s}} : \langle\, \mathit{tree}^{\mathsf{s}}, \mathit{elem}^{\mathsf{s}} \stackrel{\mathsf{s}}{\to} \mathit{bool}\rangle^{\mathsf{s}}. \\ \qquad\qquad\qquad\qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}}\mathit{find}\ \ p)\, t\ \mathsf{in}\ @^{\mathsf{d}}\mathit{balance}\,\langle\, \downarrow^{\mathit{tree}} t, \downarrow^{\mathit{elem}} x\rangle^{\mathsf{d}} \\ \qquad \mathsf{in}\ \mathit{lookup} \end{array}$$

The free variables of e_{i} are provided by e_{e} which carries all the old code and returns all the missing bindings for e_{i} to use. In this example, *balance* is the only free variable, so the result looks like:

$$\begin{array}{l} e_{\mathsf{e}} = \mathsf{let}\ \mathit{balance} = \lambda^{\mathsf{d}}\langle\, t, x\rangle^{\mathsf{d}} : \langle\, \mathit{tree}^{\mathsf{d}}, \mathit{elem}^{\mathsf{d}}\rangle^{\mathsf{d}} \ldots. \\ \qquad\qquad \mathit{lookup} \ \ = \lambda^{\mathsf{s}}\langle\, t, p\rangle^{\mathsf{s}} : \langle\, \mathit{tree}^{\mathsf{s}}, \mathit{elem}^{\mathsf{s}} \stackrel{\mathsf{s}}{\to} \mathit{bool}\rangle^{\mathsf{s}}. \\ \qquad\qquad\qquad\qquad\qquad \mathsf{let}\ x = @^{\mathsf{s}}(@^{\mathsf{s}}\mathit{find}\ \ p)\, t\ \mathsf{in}\ @^{\mathsf{d}}\mathit{balance}\,\langle\, \downarrow^{\mathit{tree}} t, \downarrow^{\mathit{elem}} x\rangle^{\mathsf{d}} \\ \qquad \mathsf{in}\ \langle\, \mathit{balance}\rangle^{\mathsf{d}} \end{array}$$

And we can combine e_{e} and e_{i} back together with $e \simeq \mathsf{let}\ \langle \mathit{fv}\rangle^{\mathsf{d}} = e_{\mathsf{e}}\ \mathsf{in}\ e_{\mathsf{i}}$ where fv is the list of free variables of e_{i}. We could of course remove *lookup* from e_{e}, but we might need it for something else (as will be shown in the next section) and it is easier to take care of it in a separate dead code elimination pass.

Because e_{e} has to return all the free variables of e_{i}, which are not known until e_{i} is complete, we cannot conveniently build e_{e} directly as we build e_{i}. Instead we build an expression with a hole E_{e} such that $e_{\mathsf{e}} \equiv E_{\mathsf{e}}[\langle \mathit{fv}\rangle^{\mathsf{d}}]$:

$$E_{\mathsf{e}} = \mathsf{let}\ \mathit{balance} = \ \ldots \quad \mathit{lookup} = \ \ldots\ \mathsf{in}\ \langle\, \bullet\rangle^{\mathsf{d}}$$

Here a TLC term with a hole E is formally defined as follows:

$$E \ ::= \ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e \ \mid\ \mathsf{let}\ x = e\ \mathsf{in}\ E$$

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$E[e]$ then fills the hole in E by textually substituting e for $\bullet$ *without* avoiding name capture:

$$(\mathsf{let}\ x = \bullet\ \mathsf{in}\ e)[e_1] \equiv (\mathsf{let}\ x = e_1\ \mathsf{in}\ e)$$
$$(\mathsf{let}\ x = e\ \mathsf{in}\ E)[e_1] \equiv (\mathsf{let}\ x = e\ \mathsf{in}\ E[e_1])$$

The splitting rules are shown in Fig. 12. The judgment $\Delta; \Gamma \vdash^{\mathsf{split}} [\![e]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}}$ states that E_{e} and e_{i} are a valid split of e in contexts Δ and Γ assuming that $e : |\sigma|^{\mathsf{s}}$. The rules only guarantee correctness of the split, but do not specify a deterministic algorithm. In practice, whenever several rules can apply, the splitting algorithm chooses the first rule shown that applies: *sp-asc* is preferred over *sp-share* which is preferred over *sp-dup*, while *sp-exp* is only applied when there is no other choice (i.e. when e is neither a let binding nor a mere variable). This ensures that we return the smallest e_{i}.

The way the rules work is as follows: *sp-asc* together with *sp-exp* turn e into a list of let bindings that ends by returning a variable; *sp-share* copies dynamic bindings to E_{e} but omits them from e_{i} while *sp-dup* copies static bindings to both E_{e} and e_{i}; finally *sp-var* replaces the terminating variable with a hole in E_{e}.

5.3.1 Splitting functions

When splitting a function f, we could apply the above algorithm to the body, and then combine the two results into two functions f_{e} and f_{i}:

$$f = \lambda^{\mathsf{s}} x : |\sigma|^{\mathsf{s}}.e \implies \begin{array}{l} f_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma|^{\mathsf{d}}.\mathsf{let}\ x = \uparrow^{\sigma} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ f_{\mathsf{i}} = \lambda^{\mathsf{s}} x : |\sigma|^{\mathsf{s}}.\mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = @^{\mathsf{d}} f_{\mathsf{e}} (\downarrow^{\sigma} x)\ \mathsf{in}\ e_{\mathsf{i}} \end{array}$$

From then on f_{i} can be used in place of f (assuming that f_{e} is in scope), so that e_{i} will be inlined without having to ever duplicate e_{e}.

As we have seen before when staging inlinable core functions, the static representation of a function is a pair of the dynamic and the static version of that function $f = \langle f_{\mathsf{d}}, f_{\mathsf{s}} \rangle^{\mathsf{s}}$. A similar representation needs to be used for module-level functions. One would be tempted to just use f_{i} for f_{s} and f_{e} for f_{d}, but a bit more work is required. First, we cannot use f_{e} directly because it only returns the free variables of f_{i} instead of the expected return value of f_{d}, but we can simply coerce f_{i} (which has f_{e} as a free variable) to a dynamic value:

$$f_{\mathsf{s}} = f_{\mathsf{i}} \qquad f_{\mathsf{d}} = \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma_1|^{\mathsf{d}}. \downarrow^{\sigma_2}(@^{\mathsf{s}} f_{\mathsf{i}} (\uparrow^{\sigma_1} x_{\mathsf{d}}))$$

The problem with this approach is that there might be some code duplication between e_{i} and e_{e}, so f_{d} might contain unnecessary copies of code already existing in f_{e}. To work around this, we slightly change the way splitting is done, so that e_{e} returns not only the free variables of e_{i}, but also the original output of e (see the *sp-var* rule):

$$E_{\mathsf{e}} = \mathsf{let}\ balance = \ \ldots \quad lookup = \ \ldots\ \mathsf{in}\ \langle \downarrow^{\sigma} lookup, \bullet \rangle^{\mathsf{d}}$$

Of course, we need to adjust f_{i} so as to select the second component of f_{e}'s result

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

to bind to its free variables. On the other hand, f_{e} is unchanged:

$$f = \lambda^{\mathsf{s}} x : |\sigma|^{\mathsf{s}}.e \implies \begin{array}{l} f_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma|^{\mathsf{d}}.\mathsf{let}\ x = \uparrow^{\sigma} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ f_{\mathsf{i}} = \lambda^{\mathsf{s}} x : |\sigma|^{\mathsf{s}}.\mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(@^{\mathsf{d}} f_{\mathsf{e}}\, (\downarrow^{\sigma} x))\ \mathsf{in}\ e_{\mathsf{i}} \end{array}$$

Since f_{e} now returns the original result in its first field, we can use it directly almost as is to build f_{d} and we can of course still use f_{i} as f_{s}:

$$f_{\mathsf{s}} = f_{\mathsf{i}} \qquad f_{\mathsf{d}} = \pi_1 \circ f_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma_1|^{\mathsf{d}}.\pi_1^{\mathsf{d}}(@^{\mathsf{d}} f_{\mathsf{e}}\, x_{\mathsf{d}})$$

5.3.2 *Properties*

To define and show correctness of the splitting algorithm, we need an extended notion of equivalence that applies to expressions rather than just values:

$$\begin{array}{c} e \simeq e' \text{ if and only if} \\ \text{for any } E \text{ such that } \cdot;\cdot \vdash E[e] : \sigma \text{ then } \cdot;\cdot \vdash E[e'] : \sigma \text{ and } \mathscr{R}_{\mathsf{e}}(E[e]) \simeq \mathscr{R}_{\mathsf{e}}(E[e']) \end{array}$$

Splitting turns an expression e into a dynamic part E_{e} and a static part e_{i}. The following theorems state that combining E_{e} and e_{i} yields a well-typed term that is semantically equivalent to e.

Theorem 5.1 (*Type preservation*)
if $\Delta;\Gamma \vdash e : |\sigma|^{\mathsf{s}}$ and $\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\, ; e_{\mathsf{i}}$ and $fv = \mathsf{fv}(e_{\mathsf{i}}) - \mathsf{dom}(\Gamma)$ then $\Delta;\Gamma \vdash \mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(E_{\mathsf{e}}[\langle fv \rangle^{\mathsf{d}}])\ \mathsf{in}\ e_{\mathsf{i}} : |\sigma|^{\mathsf{s}}$.

Theorem 5.2 (*Correctness*)
if $\Delta;\Gamma \vdash e : |\sigma|^{\mathsf{s}}$ and $\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\, ; e_{\mathsf{i}}$ and $fv = \mathsf{fv}(e_{\mathsf{i}}) - \mathsf{dom}(\Gamma)$ then $e \simeq \mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(E_{\mathsf{e}}[\langle fv \rangle^{\mathsf{d}}])\ \mathsf{in}\ e_{\mathsf{i}}$.

Both theorems can be proved via induction on the splitting derivation with the help of an invariant. For correctness, the invariant is:

$$\begin{array}{l} \text{For any term } e' \text{ and set of variables } xs \\ \text{such that } (\mathsf{fv}(e') - \mathsf{fv}(E_{\mathsf{e}}[e']) \subseteq xs \subseteq \mathsf{fv}(e') \\ \text{then } E_{\mathsf{e}}[e'] \simeq \mathsf{let}\ \langle xs \rangle^{\mathsf{d}} = \pi_2^{\mathsf{d}}(E_{\mathsf{e}}[\langle xs \rangle^{\mathsf{d}}])\ \mathsf{in}\ e' \end{array}$$

The invariant for type preservation is similar.

The splitting algorithm also satisfies the following property:

Theorem 5.3 (*Static closure*)
if $\Delta;\Gamma \vdash e : |\sigma|^{\mathsf{s}}$ and $\Delta;\Gamma \vdash^{\mathsf{split}} [\![e]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\, ; e_{\mathsf{i}}$ then all free variables in e_{i} are either bound in Γ or they have dynamic type (in the context of E_{e}).

We prove this property by inspection of the rules: if a variable is free in e_{i} but not in Γ it can only be because the *sp-share* rule was used, but that rule only applies to dynamic variables or variables which are not free in e_{i}.

Static closure implies that e_{i} contains all the inlinable sub-terms in e so splitting does not hide any inlining opportunities. In other words, when doing partial evaluation of a term containing e, we can substitute e_{i} for e without preventing any reduction (except reductions internal to the terms omitted in e_{i}, obviously). This

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$t_{\mathsf{co}} = \langle\, t_{\mathsf{d}} \xrightarrow{\mathsf{s}} t_{\mathsf{s}}, t_{\mathsf{s}} \xrightarrow{\mathsf{s}} t_{\mathsf{d}} \rangle^{\mathsf{s}}$$

$$\begin{aligned}
|\mathsf{int}|^{\mathsf{s}} &= \mathsf{int} \\
|t|^{\mathsf{s}} &= t_{\mathsf{s}} \\
|\tau_1 \to \tau_2|^{\mathsf{s}} &= \langle\, |\tau_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\tau_2|^{\mathsf{s}}, |\tau_1 \to \tau_2|^{\mathsf{d}} \rangle^{\mathsf{s}} \\
|\mathsf{V}(\tau)|^{\mathsf{s}} &= |\tau|^{\mathsf{s}} \\
|\langle\, \sigma_1, \ldots, \sigma_n \rangle|^{\mathsf{s}} &= \langle\, \langle\, |\sigma_1|^{\mathsf{s}}, \ldots, |\sigma_n|^{\mathsf{s}} \rangle^{\mathsf{s}}, |\langle\, \sigma_1, \ldots, \sigma_n \rangle|^{\mathsf{d}} \rangle^{\mathsf{s}} \\
|\sigma_1 \to \sigma_2|^{\mathsf{s}} &= \langle\, |\sigma_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\sigma_2|^{\mathsf{s}}, |\sigma_1 \to \sigma_2|^{\mathsf{d}} \rangle^{\mathsf{s}} \\
|\forall t.\sigma|^{\mathsf{s}} &= \langle\, \forall^{\mathsf{s}} t_{\mathsf{s}} : \mathsf{s}. \forall^{\mathsf{s}} t_{\mathsf{d}} : \mathsf{d}. t_{\mathsf{co}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{s}}, |\forall t.\sigma|^{\mathsf{d}} \rangle^{\mathsf{s}}
\end{aligned}
\qquad
\begin{aligned}
|\mathsf{int}|^{\mathsf{d}} &= \mathsf{int} \\
|t|^{\mathsf{d}} &= t_{\mathsf{d}} \\
|\tau_1 \to \tau_2|^{\mathsf{d}} &= |\tau_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\tau_2|^{\mathsf{d}} \\
|\mathsf{V}(\tau)|^{\mathsf{d}} &= |\tau|^{\mathsf{d}} \\
|\langle\, \sigma_1, \ldots, \sigma_n \rangle|^{\mathsf{d}} &= \langle\, |\sigma_1|^{\mathsf{d}}, \ldots, |\sigma_n|^{\mathsf{d}} \rangle^{\mathsf{d}} \\
|\sigma_1 \to \sigma_2|^{\mathsf{d}} &= |\sigma_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} |\sigma_2|^{\mathsf{d}} \\
|\forall t.\sigma|^{\mathsf{d}} &= \forall^{\mathsf{d}} t : \mathsf{d}. |\sigma|^{\mathsf{d}}
\end{aligned}$$

$$\begin{aligned}
|\,\Delta;\Gamma;\Sigma\,| \;=\; & \Delta';\Gamma' \\
& \text{where} \Delta' = \{t_{\mathsf{s}} : \mathsf{s}\,,\, t_{\mathsf{d}} : \mathsf{d} \mid t \in \mathsf{dom}(\Delta)\} \\
& \text{and} \Gamma' = \{x : |\Gamma(x)|^{\mathsf{s}} \mid x \in \mathsf{dom}(\Gamma)\} \cup \{x_t : t_{\mathsf{co}} \mid t \in \mathsf{dom}(\Delta)\} \\
& \cup \{z : |\tau|^{b} \mid \tau = \Gamma(z) \text{and} b = \Sigma(z)\}
\end{aligned}$$

Fig. 13. Type and environment translation.

in turn implies that compilation-unit boundaries have no influence on whether or not a core function gets inlined at a particular call site. We call it the *completeness* property.

5.4 The main algorithm

The main algorithm is the translation of the module language, which works similarly to (and uses) the core translation presented earlier, but is interleaved with the splitting algorithm. It also relies on the use of pairs that keep both a dynamic and a static version of every module-level value to avoid unnecessary code duplication.

Figures 13 and 14 extend the type translations $|\cdot|^{\mathsf{s}}$ and $|\cdot|^{\mathsf{d}}$ and the coercions $\downarrow$ and $\uparrow$ to the module calculus. The main change is the case for the type abstraction which we will explain later.

The full staging algorithm is shown in figure 15. The judgment $\Delta;\Gamma \vdash [\![m : \sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e$ means that under the environments Δ and Γ, the SRC module m of type σ is translated into the TLC term e. Most rules are straightforward. The translation of expressions $\iota_{\mathsf{V}}(c)$ is delegated to the core translation.

The case for module-level function and type abstraction are most interesting. The translation of a module-level function $\lambda x : \sigma.m$ begins by recursively translating the body m, and then splitting it. This is done with the judgment $\Delta;\Gamma \vdash [\![m : \sigma]\!] \Longrightarrow e_{\mathsf{e}}\,;\, E_{\mathsf{i}}$. We then build a pair $f = \langle\, f_{\mathsf{d}}, f_{\mathsf{s}} \rangle^{\mathsf{s}}$ as described in the previous section.

The translation of a type abstraction $\Lambda t.m$ follows the same pattern, except for a subtle complication introduced by typing problems discussed below.

Since all module-level code is considered static, it is tempting to think that we do not need pairing at all and can simply represent module entities with the static counterpart. But the e_{e} component obtained from λ-split is dynamic and we thus need coercions to interact with it: both the *sp-var* rule of λ-split (see figure 12) and the construction of f_{i} out of e_{i} (see section 5.3.1) introduce coercions. And since modules tend to be larger than core functions, it is even more important to avoid spurious copies.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$\downarrow^{\sigma}\colon |\sigma|^{\mathsf{s}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{d}}$ and $\uparrow^{\sigma}\colon |\sigma|^{\mathsf{d}} \xrightarrow{\mathsf{s}} |\sigma|^{\mathsf{s}}$

$$\begin{array}{ll|ll}
\downarrow^{\mathsf{int}} x & = x & \uparrow^{\mathsf{int}} x & = x \\
\downarrow^{t} x & = @^{\mathsf{s}}(\pi_2^{\mathsf{s}} x_t)\, x & \uparrow^{t} x & = @^{\mathsf{s}}(\pi_1^{\mathsf{s}} x_t)\, x \\
\downarrow^{\tau_1 \to \tau_2} x & = \pi_2^{\mathsf{s}} x & \uparrow^{\tau_1 \to \tau_2} x & = \langle\, \lambda^{\mathsf{s}} x_1 : |\tau_1|^{\mathsf{s}}.\ \uparrow^{\tau_2}(@^{\mathsf{d}} x\, (\downarrow^{\tau_1} x_1)), x \rangle^{\mathsf{s}} \\
\downarrow^{\mathsf{V}(\tau)} x & = \downarrow^{\tau} x & \uparrow^{\mathsf{V}(\tau)} x & = \uparrow^{\tau} x \\
\downarrow^{\langle \sigma_1, \ldots, \sigma_n \rangle} x & = \pi_2^{\mathsf{s}} x & \uparrow^{\langle \sigma_1, \ldots, \sigma_n \rangle} x & = \langle\, \langle\, \uparrow^{\sigma_1}(\pi_1^{\mathsf{d}} x), \ldots, \uparrow^{\sigma_n}(\pi_n^{\mathsf{d}} x) \rangle^{\mathsf{s}}, x \rangle^{\mathsf{s}} \\
\downarrow^{\sigma_1 \to \sigma_2} x & = \pi_2^{\mathsf{s}} x & \uparrow^{\sigma_1 \to \sigma_2} x & = \langle\, \lambda^{\mathsf{s}} x_1 : |\sigma_1|^{\mathsf{s}}.\ \uparrow^{\sigma_2}(@^{\mathsf{d}} x\, (\downarrow^{\sigma_1} x_1)), x \rangle^{\mathsf{s}} \\
\downarrow^{\forall t.\sigma} x & = \pi_2^{\mathsf{s}} x & \uparrow^{\forall t.\sigma} x & = \langle\, \Lambda^{\mathsf{s}} t_{\mathsf{s}} : \mathsf{s}. \Lambda^{\mathsf{s}} t_{\mathsf{d}} : \mathsf{d}. \lambda^{\mathsf{s}} x_t : t_{\mathsf{co}}.\ \uparrow^{\sigma}(x[t_{\mathsf{d}}]^{\mathsf{d}}), x \rangle^{\mathsf{s}}
\end{array}$$

Fig. 14. Binding-time coercions.

$\Delta;\Gamma \vdash [\![m : \sigma]\!] \stackrel{\mathsf{m}}{\Rightarrow} e$ such that $|\,\Delta;\Gamma;\cdot\,| \vdash e \ :\ |\sigma|^{\mathsf{s}}$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash [\![x : \Gamma(x)]\!] \stackrel{\mathsf{m}}{\Rightarrow} x} \qquad \frac{\Delta;\Gamma;\cdot \vdash [\![c : \tau]\!] \stackrel{\mathsf{s}}{\Rightarrow} e}{\Delta;\Gamma \vdash [\![\iota_{\mathsf{V}}(c) : \mathsf{V}(\tau)]\!] \stackrel{\mathsf{m}}{\Rightarrow} e}$$

$$\frac{\Delta;\Gamma \vdash x_i \ :\ \sigma_i \ \ (1 \leqslant i \leqslant n) \qquad \sigma = \langle\, \sigma_1, \ldots, \sigma_n \rangle}{\Delta;\Gamma \vdash [\![\langle\, x_1, \ldots, x_n \rangle \ :\ \sigma]\!] \stackrel{\mathsf{m}}{\Rightarrow} \begin{array}{l} \mathsf{let}\ x_{\mathsf{s}} = \langle\, x_1, \ldots, x_n \rangle^{\mathsf{s}} \\ \quad\ x_{\mathsf{d}} = \langle\, \downarrow^{\sigma_1} x_1, \ldots, \downarrow^{\sigma_n} x_n \rangle^{\mathsf{d}} \\ \mathsf{in}\ \langle\, x_{\mathsf{s}}, x_{\mathsf{d}} \rangle^{\mathsf{s}} \end{array}}$$

$$\frac{\Delta;\Gamma \vdash x \ :\ \langle\, \sigma_1, \ldots, \sigma_n \rangle \quad 1 \leqslant i \leqslant n}{\Delta;\Gamma \vdash [\![\pi_i x : \sigma_i]\!] \stackrel{\mathsf{m}}{\Rightarrow} \pi_i^{\mathsf{s}}(\pi_1^{\mathsf{s}} x)} \qquad \frac{\Delta;\Gamma \vdash x_1 \ :\ \sigma_2 \to \sigma_1 \quad \Delta;\Gamma \vdash x_2 \ :\ \sigma_2}{\Delta;\Gamma \vdash [\![@ x_1\, x_2 : \sigma_2]\!] \stackrel{\mathsf{m}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} x_1)\, x_2}$$

$$\frac{\Delta;\Gamma, x : \sigma_1 \vdash [\![m : \sigma_2]\!] \Longrightarrow e_{\mathsf{e}}\ ;\ E_{\mathsf{i}}}{\begin{array}{l} \Delta;\Gamma \vdash [\![\lambda x : \sigma_1 . m : \sigma_1 \to \sigma_2]\!] \stackrel{\mathsf{m}}{\Rightarrow} \\ \mathsf{let}\ x_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma_1|^{\mathsf{d}}.\ \mathsf{let}\ x = \uparrow^{\sigma_1} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ \quad\ x_{\mathsf{i}} = \lambda^{\mathsf{s}} x : |\sigma_1|^{\mathsf{s}}.\ E_{\mathsf{i}}[\pi_2^{\mathsf{d}}(@^{\mathsf{d}} x_{\mathsf{e}}\, (\downarrow^{\sigma_1} x))] \\ \mathsf{in}\ \langle\, x_{\mathsf{i}}, \lambda^{\mathsf{d}} x_{\mathsf{d}} : |\sigma_1|^{\mathsf{d}}. \pi_1^{\mathsf{d}}(@^{\mathsf{d}} x_{\mathsf{e}}\, x_{\mathsf{d}}) \rangle^{\mathsf{s}} \end{array}}$$

$$\frac{\Delta, t;\Gamma \vdash [\![m : \sigma]\!] \Longrightarrow e_{\mathsf{e}}\ ;\ E_{\mathsf{i}}}{\begin{array}{l} \Delta;\Gamma \vdash [\![\Lambda t.m : \forall t.\sigma]\!] \stackrel{\mathsf{m}}{\Rightarrow} \\ \mathsf{let}\ x_{\mathsf{e}} = \Lambda^{\mathsf{d}} t : \mathsf{d}.\ \{t/t_{\mathsf{d}}, t/t_{\mathsf{s}}, \langle\, \mathsf{id}_t^{\mathsf{s}}, \mathsf{id}_t^{\mathsf{s}} \rangle^{\mathsf{s}}/x_t\} e_{\mathsf{e}} \\ \quad\ x_{\mathsf{i}} = \Lambda^{\mathsf{s}} t_{\mathsf{s}} : \mathsf{s}. \Lambda^{\mathsf{s}} t_{\mathsf{d}} : \mathsf{d}. \lambda^{\mathsf{s}} x_t : t_{\mathsf{co}}.\ E_{\mathsf{i}}[\pi_2^{\mathsf{d}}(x_{\mathsf{e}}[t_{\mathsf{d}}]^{\mathsf{d}})] \\ \mathsf{in}\ \langle\, x_{\mathsf{i}}, \Lambda^{\mathsf{d}} t : \mathsf{d}. \pi_1^{\mathsf{d}}(x_{\mathsf{e}}[t]^{\mathsf{d}}) \rangle^{\mathsf{s}} \end{array}}$$

$$\frac{\Delta;\Gamma \vdash x \ :\ \forall t.\sigma}{\Delta;\Gamma \vdash [\![x[\tau] \ :\ \{\tau/t\}\sigma]\!] \stackrel{\mathsf{m}}{\Rightarrow} @^{\mathsf{s}}(\pi_1^{\mathsf{s}} x)[|\tau|^{\mathsf{s}}]^{\mathsf{s}}[|\tau|^{\mathsf{d}}]^{\mathsf{s}} \langle\, \downarrow^{\tau}, \uparrow^{\tau} \rangle}$$

$$\frac{\Delta;\Gamma \vdash [\![m_1 : \sigma_1]\!] \stackrel{\mathsf{m}}{\Rightarrow} e_1 \quad \Delta;\Gamma, x : \sigma_1 \vdash [\![m_2 : \sigma_2]\!] \stackrel{\mathsf{m}}{\Rightarrow} e_2}{\Delta;\Gamma \vdash [\![\mathsf{let}\ x = m_1\ \mathsf{in}\ m_2 : \sigma_2]\!] \stackrel{\mathsf{m}}{\Rightarrow} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}$$

$\Delta;\Gamma \vdash [\![m : \sigma]\!] \Longrightarrow e_{\mathsf{e}}\ ;\ E_{\mathsf{i}}$
such that $|\,\Delta;\Gamma;\cdot\,| \vdash E_{\mathsf{i}}[\pi_2^{\mathsf{d}} e_{\mathsf{e}}] \ :\ |\sigma|^{\mathsf{s}}$

$$\frac{\begin{array}{ll} \Delta;\Gamma \vdash [\![m : \sigma]\!] \stackrel{\mathsf{m}}{\Rightarrow} e & |\,\Delta;\Gamma;\cdot\,| \vdash^{\mathsf{split}} [\![e]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}} \\ fv = \mathsf{fv}(e_{\mathsf{i}}) - \mathsf{dom}(\Gamma) & E_{\mathsf{i}} = (\mathsf{let}\ \langle fv \rangle^{\mathsf{d}} = \bullet\ \mathsf{in}\ e_{\mathsf{i}}) \end{array}}{\Delta;\Gamma \vdash [\![m : \sigma]\!] \Longrightarrow E_{\mathsf{e}}[\langle fv \rangle^{\mathsf{d}}]\ ;\ E_{\mathsf{i}}}$$

Fig. 15. Module code translation. id_t^b is a shorthand for $\lambda^b x : t.x$.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

Type abstractions As mentioned above, type abstractions introduce some complications. The problem appears when we try to define coercion functions. The naive approach would look like:

$$\downarrow^{\forall t.\sigma} x = \Lambda^{\mathsf{d}} t : \mathsf{d}.\ \downarrow^{\sigma}(x[t]^{\mathsf{s}})$$
$$\uparrow^{\forall t.\sigma} x = \Lambda^{\mathsf{s}} t : \mathsf{s}.\ \uparrow^{\sigma}(x[t]^{\mathsf{d}})$$

but this is not type correct, since t in the second rule can be static and hence cannot be passed to the dynamic x. Obviously, we need here the same kind of (contra-variant) argument coercion as we use on functions, but our language does not provide us with any way to create a $\downarrow$ operator to apply to types.

Furthermore, the two inner coercions $\downarrow^{\sigma}$ and $\uparrow^{\sigma}$ are not very well defined since σ can have a free variable t. This begs the question: what should $\uparrow^{t}$ do ?

There are several ways to solve these two problems:

- Give up on static type arguments and force any type-variable to be dynamic. This restriction is fairly minor in practice. It only manifests itself when a function is manipulated as data by polymorphic code, such as when a function is passed to the identity function id: the function returned by $@\,\mathsf{id}\,f$ cannot be inlined even if f is.
- Extend our language with a more powerful type-system that allows intensional type-analysis (Harper & Morrisett, 1995). This seems possible, but would complicate the type-system considerably and potentially the staging and the coercions as well.
- Use a dictionary-passing approach (Wadler & Blott, 1989): instead of trying to coerce our static t into a dynamic t, we can simply always provide both versions t_{s} and t_{d} along with both $\downarrow^{t}$and $\uparrow^{t}$so that the coercions are constructed at the type application site, where t is statically known.

The first solution is simple and effective, but we opted for the third alternative because it has fewer limitations. The static version of $\Lambda t.m$ (before pairing with its dynamic counterpart) looks like:

$$\Lambda^{\mathsf{s}} t_{\mathsf{s}} : \mathsf{s}. \Lambda^{\mathsf{s}} t_{\mathsf{d}} : \mathsf{d}. \lambda^{\mathsf{s}} x_t : t_{\mathsf{co}}. e$$

This means that for every t in the SRC Δ environment, we now have two corresponding type variables t_{s} and t_{d} plus one value variable x_t which holds the two coercion functions $\downarrow^{t}$ and $\uparrow^{t}$ As can be seen in Fig. 13 (which refines figure 10) where t_{co} is also defined. This notation is used for convenience in all the figures.

Such an encoding might look convoluted and cumbersome, but type abstractions only represent a small fraction of the total code size and the run-time code size is unaffected, so it is a small price to pay in exchange for the ability to inline code that had to pass through a function like id.

Theorem 5.4 (*Type preservation*)
If $\Delta;\Gamma \vdash m : \sigma$ and $\Delta;\Gamma \vdash [\![m : \sigma]\!] \Longrightarrow e_{\mathsf{e}} ; E_{\mathsf{i}}$ then $|\Delta;\Gamma;\cdot| \vdash E_{\mathsf{i}}[\pi_2^{\mathsf{d}} e_{\mathsf{e}}] : |\sigma|^{\mathsf{s}}$.

Together with the residualization theorem 4.1, this means that after specialization all the inlinable code has been inlined away.

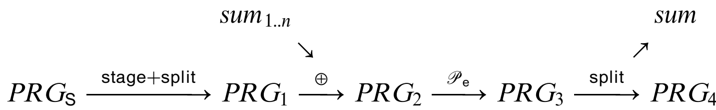
https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

6 Handling the top level

The above presentation only explains how to translate each SRC compilation unit into its TLC counterpart. This section describes in detail how to handle top-level issues to link multiple compilation units together.

Handling of compilation units is not difficult, but is worth looking at not only to get a better idea of how the code flows through the compiler, but also because the treatment of side-effects depends on the specifics of the evaluation of each compilation unit (see section 7.1).

As can be seen in figure 1, we apply λ-split twice. This derives from the need to handle the top-level of the compilation unit in a special way where splitting internal module functions should be done early, while splitting the top-level should be done late. Here is a slightly more detailed diagram:



Instead of spreading the split into two parts, we could of course do it once and for all at the very beginning, but then we would lose the opportunity to move into the export summary copies of wrapper functions (used e.g. for uncurrying, unboxing or flattening) introduced by the intermediate optimization phases.

Doing the split in two steps also forces us to apply the split to TLC terms (it would be silly to have two splitting algorithms). This also motivates our choice to interleave the staging and splitting since the splitting algorithm needs to know which functions are modules and which are not because splitting core functions is often detrimental to performance.

The top-level also gets a special treatment because of separate compilation. A compilation-unit can contain free variables, which are essentially the imports of the unit. Instead of considering such an open term, we close it by turning it into a function from its imports to its exports. More specifically, a compilation-unit in SRC will look like:

$$PRG_{\mathsf{S}} = \lambda\langle\, imp_1, \ldots, imp_n \rangle : \langle\, \sigma_1, \ldots, \sigma_n \rangle.m$$

The translation to TLC assumes that the function is static as well as the imports (these will be import summaries, which are by essence inlinable, after all) and simply translates the body using:

$$\cdot; \{imp_i : |\sigma_i|^{\mathsf{s}} \mid 1 \leqslant i \leqslant n\} \vdash [\![m : \sigma]\!] \overset{\mathsf{m}}{\Rightarrow} e$$

This recursively splits each and every internal module-level function (the recursion is done by the staging part of the translation which calls λ-split when needed, see the rules for λ and Λ in figure 15), but leaves the top-level function alone. These internal splits are necessary to allow inlining across module boundaries but still within a compilation unit. The program now looks like:

$$PRG_1 = \lambda^{\mathsf{s}}\langle\, imp_1, \ldots, imp_n \rangle^{\mathsf{s}} : \langle\, |\sigma_1|^{\mathsf{s}}, \ldots, |\sigma_n|^{\mathsf{s}} \rangle^{\mathsf{s}}.e$$

The next step is to bring in copies of the import summaries. Every imp_i has a

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

corresponding summary sum_i generated when that import was compiled. Summaries are the e_{i} half of a split and thus contain free dynamic variables. So we replace each imp_i argument with a copy of sum_i, and add the corresponding new free variables imp_{ij} as new arguments:

$$\begin{aligned} PRG_2 = \lambda^{\mathsf{s}}\langle\, imp_{11},\ldots,imp_{nk}\rangle^{\mathsf{d}} : \langle\, \sigma_{11},\ldots,\sigma_{nk}\rangle^{\mathsf{d}}.\\ \mathsf{let}\ imp_1 = sum_1 \\ \cdots \\ imp_n = sum_n \\ \mathsf{in}\ e \end{aligned}$$

After that comes the actual partial-evaluation and optimization which ends with a term PRG_3 very much like PRG_2 but with an optimized body e_{o} exempt of any static redex:

$$PRG_3 = \mathscr{P}_{\mathsf{e}}(PRG_2) = \lambda^{\mathsf{s}}\langle\, imp_{11},\ldots,imp_{nk}\rangle^{\mathsf{d}} : \langle\, \sigma_{11},\ldots,\sigma_{nk}\rangle^{\mathsf{d}}.\, e_o$$

We then pass it to the second λ-split, along with the SRC σ output type that we remembered from the staging phase:

$$\Delta;\Gamma \vdash^{\mathsf{split}} [\![e_{\mathsf{o}}]\!] \stackrel{\sigma}{\Longrightarrow} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}}$$

This split gives us a residual program and an export summary sum that will be used as a sum_i next time around:

$$\begin{aligned} sum &= \lambda^{\mathsf{s}}\langle fv\rangle^{\mathsf{d}} : \langle\, \sigma_{fv}\rangle^{\mathsf{d}}.e_{\mathsf{i}} \qquad\qquad \text{where} fv = \mathsf{fv}(e_{\mathsf{i}}) \\ PRG_4 &= \lambda^{\mathsf{d}}\langle\, imp_{11},\ldots,imp_{nk}\rangle^{\mathsf{d}} : \langle\, \sigma_{11},\ldots,\sigma_{nk}\rangle^{\mathsf{d}}.E_{\mathsf{e}}[\langle fv\rangle^{\mathsf{d}}] \end{aligned}$$

The export summary sum will be stashed somewhere to be used when a compilation unit wants to import it. As for PRG_4, it continues through the remaining compilation stages down to machine code.

When the program is run, all the compilation units need to be instantiated in the proper order. Once all the imports imp_{ij} of our unit have been built, PRG_4 is run as follows:

$$\langle\, exp, \langle fv\rangle^{\mathsf{d}}\rangle^{\mathsf{d}} = \mathscr{R}_{\mathsf{e}}(@^{\mathsf{d}} PRG_4\, \langle\, imp_{11},\ldots,imp_{nk}\rangle^{\mathsf{d}})$$

Here exp is the original exports of this compilation unit. They will be ignored for all but the main compilation unit (unless one of the dependent units was compiled without cross-module inlining, in which case exp will be used by that unit). $\langle fv\rangle^{\mathsf{d}}$ is the set of exports generated by the λ-split and needed for all the compilation units that depend on the current unit and hence imported sum. When running those dependent units, the current $\langle fv\rangle^{\mathsf{d}}$ will then appear as the arguments $imp_{i1}\ldots imp_{ik}$.

Note that PRG_4 will only be run once and for all whereas sum will be evaluated as many times as it is imported by dependent compilation units. Also, PRG_4 is not completely dynamic since the coercion $\downarrow^{\sigma} x$ of the *sp-var* rule (in figure 12) introduces static redexes; we have to perform another round of partial evaluation on PRG_4 before feeding it to the backend code generator.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

$$
\begin{array}{llcl}
(\mathit{type}) & \sigma & ::= & \ldots \mid \exists^b t:k.\sigma \\
(\mathit{term}) & e & ::= & \ldots \mid \mathsf{open}^b\ v\ \mathsf{as}\ (t,x)\ \mathsf{in}\ e \\
(\mathit{value}) & v & ::= & \ldots \mid \mathsf{pack}^b(t=\sigma:k, e)
\end{array}
$$

$$
\begin{array}{lcl}
|\sigma_1 \to \sigma_2|^{\mathsf{s}} & = & \exists^{\mathsf{s}} t:\mathsf{d}.\langle\, |\sigma_1|^{\mathsf{s}} \xrightarrow{\mathsf{s}} t \xrightarrow{\mathsf{s}} |\sigma_2|^{\mathsf{s}}, |\sigma_1|^{\mathsf{d}} \xrightarrow{\mathsf{d}} \langle\, |\sigma_2|^{\mathsf{d}}, t\rangle^{\mathsf{d}}\rangle^{\mathsf{s}} \\
\downarrow^{\sigma_1\to\sigma_2} x & = & \lambda^{\mathsf{d}} x_1:|\sigma_1|^{\mathsf{d}}.\mathsf{open}^{\mathsf{s}}\ x\ \mathsf{as}\ (t,x)\ \mathsf{in}\ \pi_1^{\mathsf{d}}(@^{\mathsf{d}}(\pi_2^{\mathsf{s}}x)\, x_1) \\
\uparrow^{\sigma_1\to\sigma_2} x & = & \mathsf{pack}^{\mathsf{s}}(t=\langle\,\rangle^{\mathsf{d}}:\mathsf{d}, \\
 & & \quad \mathsf{let}\ x_{\mathsf{s}} = \lambda^{\mathsf{s}} x_1:|\sigma_1|^{\mathsf{s}}.\lambda^{\mathsf{s}} x_2:t.\ \uparrow^{\sigma_2}(@^{\mathsf{d}} x\ x_1) \\
 & & \quad\quad x_{\mathsf{d}} = \lambda^{\mathsf{d}} x_3:|\sigma_1|^{\mathsf{d}}.\langle\, @^{\mathsf{d}} x\ x_3, \langle\,\rangle^{\mathsf{d}}\rangle^{\mathsf{d}} \\
 & & \quad \mathsf{in}\ \langle\, x_{\mathsf{s}}, x_{\mathsf{d}}\rangle^{\mathsf{s}}\ )
\end{array}
$$

$$
\frac{\Delta;\Gamma \vdash x_1 : \sigma_2 \to \sigma_1 \quad \Delta;\Gamma \vdash x_2 : \sigma_2}{\begin{array}{l}\Delta;\Gamma \vdash [\![@x_1\, x_2 : \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{open}^{\mathsf{s}}\ x_1\ \mathsf{as}\ (t,x) \\ \mathsf{in\ let}\ x_{\mathsf{e}} = @^{\mathsf{d}}(\pi_2^{\mathsf{s}}x)(\downarrow^{\sigma_1} x_2) \\ \quad \mathsf{in}\ @^{\mathsf{s}}(@^{\mathsf{s}}(\pi_1^{\mathsf{s}}x)\, x_2)(\pi_2^{\mathsf{d}} x_{\mathsf{e}})\end{array}}
$$

$$
\frac{\Delta;\Gamma, x:\sigma_1 \vdash [\![m : \sigma_2]\!] \Longrightarrow e_{\mathsf{e}}\ ;\ E_{\mathsf{i}}}{\begin{array}{l}\Delta;\Gamma \vdash [\![\lambda x:\sigma_1.m : \sigma_1 \to \sigma_2]\!] \overset{\mathsf{m}}{\Rightarrow} \\ \mathsf{pack}^{\mathsf{s}}(t=\sigma_{fv}:\mathsf{d}, \\ \quad \mathsf{let}\ x_{\mathsf{e}} = \lambda^{\mathsf{d}} x_{\mathsf{d}}:|\sigma_1|^{\mathsf{d}}.\ \mathsf{let}\ x = \uparrow^{\sigma_1} x_{\mathsf{d}}\ \mathsf{in}\ e_{\mathsf{e}} \\ \quad\quad x_{\mathsf{i}} = \lambda^{\mathsf{s}} x:|\sigma_1|^{\mathsf{s}}.\lambda^{\mathsf{s}} y:t.\ E_{\mathsf{i}}[y] \\ \quad \mathsf{in}\ \langle\, x_{\mathsf{i}}, x_{\mathsf{e}}\rangle^{\mathsf{s}}\ )\end{array}}
$$

Fig. 16. New rules using existential types.
The type, kind, and evaluation semantics should be extended correspondingly. Furthermore, the last rule needs the type σ_{fv} of the free variables of e_{i} which can easily be propagated during splitting.

7 Extensions

In order to model real world inliners faithfully, our translation still needs various additions which we have not explored in depth yet. We present some here along with other potential extensions.

7.1 Side effects

Introducing side-effects is mostly straightforward, with just one exception: The *sp-dup* rule in figure 12 cannot be applied to non-pure terms since it would duplicate their effects. But reverting to the *sp-share* rule (also in figure 12) for those side-effecting terms is not an option either because we would then lose the completeness property that we are looking for. An alternative is to use the following *sp-move* rule that moves the binding to e_{i} instead of merely copying it:

$$
\frac{\Delta;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta;\Gamma, x:\sigma_1 \vdash^{\mathsf{split}} [\![e_2]\!] \overset{\sigma}{\Longrightarrow} E_{\mathsf{e}}\ ;\ e_{\mathsf{i}}}{\begin{array}{c}\Delta;\Gamma \vdash^{\mathsf{split}} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] \overset{\sigma}{\Longrightarrow} \\ \mathsf{let}\ x_1 = \lambda^{\mathsf{d}} x:\sigma_1.E_{\mathsf{e}}[\langle fv\rangle^{\mathsf{d}}]\ \mathsf{in}\ \bullet\ ; \\ \mathsf{let}\ x = e_1\ \mathsf{in\ let}\ \langle fv\rangle^{\mathsf{d}} = @^{\mathsf{d}} x_1\,(\downarrow^{\sigma_1} x)\ \mathsf{in}\ e_{\mathsf{i}}\end{array}}
$$

As presented, this rule is not quite correct because the coercion $\downarrow^{\sigma_1}$ assumes σ_1 is an SRC type but σ_1 is really a TLC type. This can be easily resolved by passing all the SRC types during the translation. Alternatively, we could represent $|\mathsf{int}|^{\mathsf{s}}$ as $\langle\, \mathsf{int}, \mathsf{int}\rangle^{\mathsf{s}}$, then the coercion $\downarrow$ simply becomes π_2^{s} for all types.

A potential issue is that, as can be seen in section 6, the top-level E_{e} is evaluated

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

once and for all in a global environment, while e_i will be evaluated each time it is imported into a client unit. This means that the top-level e_i must be free of side-effects. Luckily, we can show that this problem does not appear: the compilation unit does not contain any static free variables or static redexes; so the only static code to split into e_i is composed exclusively of values, which have no side effects.

This *sp-move* rule could also be used for other purposes, such as specializing a binding in the context of the client, as was suggested briefly in Blume and Appel's paper (they did not have such a rule).

Another approach altogether to the handling of side-effects is to notice that since e_i has to be pure, we can turn $@^\mathsf{s} x_1\, x_2$ into $@^\mathsf{d}(@^\mathsf{s} x_\mathsf{i}\, x_2)\,(\pi_2^\mathsf{d}(@^\mathsf{d} x_\mathsf{e}\,(\downarrow x_2)))$ and then split out the remaining static application which is known to be pure. We do not even need to change the splitting algorithm itself, but just two rules in figure 15.

The key is that we can do this rewrite even if we do not know x_1. We simply need to represent x_1 as a pair $\langle x_\mathsf{i}, x_\mathsf{e}\rangle^\mathsf{s}$. But of course, this is already the case for other reasons, so the changes are very minor.

Of course, there is a catch: the type of the free variables of e_i suddenly leak into the type of $|\sigma_1 \to \sigma_2|^\mathsf{s}$ which becomes an existential type.

Figure 16 shows what the rules would look like. Some of the work is now shifted from the function definition to the function application, but overall, the complexity of terms is not seriously impacted. Apart from the introduction of existential types, this translation variant also requires an impredicative calculus. TLC was already impredicative, so no changes were required there.

7.2 Recursion

The TLC calculus lacks fixpoint. Adding recursive functions does not pose any conceptual problem, except for the risk of compilation not terminating. There are several reasonable solutions to this problem either from the inlining community or from the partial evaluation community. The most trivial solution is to allow fixpoint on dynamic terms only, which amounts to disallowing inlining of recursive functions, but since it can be important to allow inlining even in the presence of recursion, one can also do a little bit of analysis to find a conservative estimate of whether or not a risk of infinite recursion is present (Peyton Jones & Marlow, 1999).

Recursion on types is more challenging since recursive data-structures cannot (or should not) be coerced. In our case, however, this restriction only applies to coercions from d to s, so we can always work around the problematic cases by forcing recursive data-structures to be dynamic.

7.3 Optimizations as staged computation

With a full λ-calculus available at compile-time, we can now provide facilities similar to macros, or rather to Lisp's compiler macros. For example, a compiler macro for multiplication could test its arguments at compile time and replace the multiplication with some other operations depending on whether or not one of the value is statically known and what value it takes. Using such a facility we could

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

move some of the optimizations built into the compiler into a simple library, making them easily extensible.

A more realistic use in the short term is to encode the predicate for conditional inlining directly into the language. The current inliner allows inlining hints more subtle than the ones present in SRC. They can express a set of conditions that should hold at the call site in order for the call to be inlined. For example, `map` will only be inlined if it is applied to a known function.

We could now strip out those ad-hoc annotations and simply write `map` as a compile-time function that intensionally analyzes its arguments and either returns a copy of its body if the function argument is a λ-expression or returns just a call to the common version if the argument is a variable (i.e. an escaping function).

7.4 Staging refinement

Partial evaluation as well as other optimizations will sufficiently change the shape of the code to justify or even require refining the binding-time annotations. This can happen because a function has been optimized down to just a handful of statements, or because it has been split into a wrapper and a main body or any other reason.

Turning a dynamic function into a static one is not very difficult to implement, but more work needs to be done to express it cleanly within our framework.

It seems to require among other things the ability to optimize away pairs of coercions that cancel each other out such as not only $\downarrow^{\sigma}\uparrow^{\sigma}x$ (which is trivially done by the partial evaluator) but also $\uparrow^{\sigma}\downarrow^{\sigma}x$ which appears to involve evaluation of dynamic code at compile-time.

7.5 Link-time optimizations

Another extension is to add multiple levels so that we can express compile-time execution, link-time execution, run-time code generation and more.

This will require extending a calculus such as $\lambda^{\bigcirc}$ (Davies, 1996) with at least some form of polymorphism, but should not pose any real problem, except that the kind of tricks we used to work around the lack of simple coercion for type abstraction might need to be generalized to n-levels. If n is unbounded, it might not be possible and even if it is bounded, it might be impractical.

Also, the use of pairs of fully-static and fully-dynamic representations of the same original expressions would not generalize to n-levels easily, but could still be kept for the benefit of the compile stage.

7.6 Implementation

As mentioned in the introduction, this paper was motivated by the need to better understand the behavior of our inliner in SML/NJ. Since our implementation handles the complete SML language in a production compiler, it has to deal with all the issues mentioned above. Here are the most important differences between the model presented in this paper and the actual code:

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

- For historical reasons, our intermediate language is predicative, which prevents us from using existential packages to solve the problem of side effects. Instead we simply revert to using the *sp-share* rule and lose the completeness property.
- Our language allows recursion both for dynamic and for static functions. Termination of $\mathscr{P}_{\mathsf{e}}(e)$ is ensured by a simple conservative loop detection.
- The dynamic and static pairs we use to avoid spurious code duplication are represented in an ad-hoc way that eliminates the redundancy. This ad-hoc representation looked like a good idea at the time, but made it unnecessarily painful to add the refinement described in section 5.3.1.

8 Related work

Functional-language compilers such as O'Caml (Leroy, 1995), SML/NJ (Appel, 1991), GHC (Peyton Jones & Marlow, 1999) and TIL (Tarditi, 1996) all spend great efforts to provide better support to inlining. Although none of them models inlining as staged computation, the heuristics for detecting what functions should be inlined are still useful in our framework. In fact, our FLINT optimizer (Monnier *et al.*, 1999) inherits most of the heuristics used in the original SML/NJ compiler.

Control-Flow Analysis (CFA) (Shivers, 1991; Ashley, 1997) is an alternative to λ-splitting to propagate inlining information across functions and functors. It tries to find, for example via abstract interpretation, the set of functions possibly invoked at each call site in the program. It offers the advantage of requiring less code duplication and may expose more opportunities for inlining inside a compilation unit. For example, in a code such as:

$$\mathsf{let}\ f\ x\ y = ..y\ x..\ \mathsf{and}\ g\ x = ...\ \mathsf{in}\ \langle f\ 1\ g, f\ 2\ g\rangle$$

CFA can inline the function g into f without inlining f whereas our inliner will only reach the same result if it can first inline the two calls to f. On the other hand, in a code such as $f\ g\ x$ where f is a functor that ends up returning its argument unchanged, our inliner will be able to replace the code with $g\ x$, no matter how f is defined, whereas in the case of CFA, if the definition of f is sufficiently complex, a costly polyvariant analysis is needed to discover that the code can be replaced with $g\ x$.

Partial evaluation is a very active research area. Jones *et al.* (1993) gives a good summary about some of the earlier results. Danvy's paper (1996) on type-directed partial evaluation inspired us to look into sophisticated forms of binding-time coercions.

Tempo (Consel & Noël, 1996; Marlet *et al.*, 1999) is a C compiler that makes extensive use of partial-evaluation technologies. Its main emphasis is however on efficient runtime code generation. Sperber & Thiemann (1996, 1997) worked on combining compilation with partial evaluation, however they were not concerned with modeling the inlining optimization as done in a production compiler.

Nielson & Nielson (1992) gave an introduction to a two-level λ-calculus. Davies & Pfenning (1996) proposed to use modal logic to express staged computation. Moggi (1997) pointed out that both of these calculi are subtly different from the

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

two-level calculus used in partial evaluation (Jones *et al*., 1993). Taha *et al*. (Taha, 1999; Taha & Sheard, 1997; Moggi *et al*., 1999) showed how to combine these different calculi into a single framework.

Our TLC calculus (see section 4) is an extension of Moggi's two-level $\lambda 2sd$ calculus (Moggi, 1997) with the System-F-style polymorphism (Girard, 1972; Reynolds, 1974). Davies (1996) used the temporal logic to model an n-level calculus which naturally extends $\lambda 2sd$.

Foster *et al*. (1999) proposed to use qualified types to model source-level program directives. Their framework can be applied to binding time annotations but these annotations would have to become parts of the type specifications. Our inlining annotations on the other hand do not change the source-level type specifications.

Blume & Appel (1997) suggested λ-splitting to support cross module inlining. Their algorithm is based on a weakly-typed λ-calculus and provides a convenient cross-module inlining algorithm. Our work extends theirs by porting their algorithm to a much more powerful language and formalizing it. By using the two-level λ-calculus we can express some of the inliner's behavior in the types.

O'Caml (Leroy, 1995) collects the small inlinable functions of a module into its *approximation* and then reads in this extra info (if available) when compiling a client module. It works very well across modules and can even inline functions from within a functor to the client of the functor, but is unable to inline the argument of a functor. For example, passing a module through a trivial 'adaptor' functor (which massages a module to adapt it to some other signature, for example) will lose the *approximation*, preventing inlining.

By encoding the equivalent of *approximations* directly into types, Shao (1998) presents an alternative approach which allows the full inlining information to be completely propagated across functor applications by propagating it along with the types. But this comes at the cost of a further complication of the module elaboration. Another problem is that some of the functions we might want to inline (such as uncurry wrappers) do not yet exist at the time of module elaboration.

Recently, Ganz *et al*. (2001) presented an expressive, typed language that supports generative macros. The language, MacroML, is defined by an interpretation into MetaML (Taha & Sheard, 1997). This is similar to our approach because macros can also be viewed as inlinable functions; the translation from MacroML to MetaML resembles our translation from the source calculus SRC to the target calculus TLC. There are, however, several major differences. First, in MacroML, macros and functions are different language constructs; macros never escape so they can be unconditionally marked as 'static' and no coercions or polymorphic binding-time annotations are ever needed; in our SRC calculus, however, functions that are marked as inlinable are still treated as regular functions so they can escape in any way they like. Secondly, it is unclear how MacroML can be extended to support ML-style modules; MacroML assigns different types to macros and functions so exporting macros would require adding new forms of specifications into ML signatures; in our SRC language, however, a function is always assigned the same type whether it is marked as inlinable or not, so we can just reuse the existing ML module language. Thirdly, unlike MacroML, we presented various

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

techniques to control code duplication – this is cruicial for cross-module inlining since naively expanding every functor in ML would certainly cause code explosion.

9 Conclusions and future work

Expressing inlining in terms of a staged computation allows us to better formalize the behavior of the inliner and provide strong guarantees of what gets inlined where.

We have shown how this can be done in the context of a realistic two-level polymorphic language and how it interacts with cross-module inlining. The formalism led us to a clean design in which we can easily show that code is only and always duplicated when useful.

The present design eliminates run-time penalties usually imposed by the powerful abstraction mechanism offered by parameterized modules, enabling a more natural programming style. More importantly, our algorithm provides such flexibility while still maintaining separate compilation.

An interesting question is whether or not using monomorphic staging annotations was a good choice. It seems that polymorphism could allow us to do away with the coercions, although it would at least require the use of continuation passing style in order to maintain precision of annotations.

Acknowledgements

We would like to thank the anonymous referees as well as Dominik Madon, Chris League, and Walid Taha for their comments and suggestions on an early version of this paper. This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title 'Scaling Proof-Carrying Code to Production Compilers and Security Policies,' ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9633390 and CCR-9901011. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

References

Appel, A. W. (1991) *Compiling with Continuations.* Cambridge University Press.

Appel, A. W. and MacQueen, D. B. (1991) Standard ML of New Jersey. In: Wirsing, M., editor, *Third Int. Symp. on Prog. Lang. Implementation and Logic Programming*, pp. 1–13. New York: Springer-Verlag.

Asai, K. (1999) Binding-time analysis for both static and dynamic expressions. *Static Analysis Symposium*, pp. 117–133.

Ashley, M. J. (1997) The effectiveness of flow analysis for inlining. *Int. Conf. on Functional Programming*. ACM Press.

Birkedal, L. and Welinder, M. (1995) Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, **8**(3), 191–208.

Blume, M. (1995) *Standard ML of New Jersey compilation manager.* Manual accompanying SML/NJ software.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

Blume, Ma. and Appel, A. W. (1997) Lambda-splitting: A higher-order approach to cross-module optimizations. *Int. Conf. on Functional Programming*. ACM Press.

Consel, C. (1993) Polyvariant binding-time analysis for applicative languages. *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 145–154.

Consel, C. and Noël, F. (1996) A general approach for run-time specialization, and its application to C. *Symposium on Principles of Programming Langueges*. ACM Press.

Danvy, O. (1996) Type-directed partial evaluation. *Symposium on Principles of Programming Langueges*. ACM Press.

Danvy, O.r, Malmkjær, K. and Palsberg, J. (1996) Eta-expansion does The Trick. *IEEE Trans. Program. Lang. Syst.* **8**(6), 730–751.

Davies, R. (1996) A temporal-logic approach to binding-time analysis. *11th Annual Symposium on Logic in Computer Science*, pp. 184–195.

Davies, R. and Pfenning, F. (1996) A modal analysis of staged computation. *Symposium on Principles of Programming Langueges*. ACM Press.

Flanagan, C., Sabry, A., Duba, B. F. and Felleisen, M. (1993) The essence of compiling with continuations. *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pp. 237–247. ACM Press.

Foster, J. S., Fähndrich, M. and Aiken, A. (1999) A theory of qualified types. *Symposium on Programming Languages Design and Implementation*. ACM Press.

Ganz, S., Sabry, A. and Taha, W. (2001) Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. *International Conference on Functional Programming*. ACM Press.

Girard, J. Y. (1972) *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII.

Glynn, K., Stuckey, P. J., Sulzmann, M. and Søndergaard, H. (2001) Boolean constraints for binding-time analysis. *Program as Data Objects*, pp. 39–62.

Harper, B. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. *Symposium on Principles of Programming Languages*, pp. 130–141.

Harper, R., Mitchell, J. C. and Moggi, E. (1990) Higher-order modules and the phase distinction. *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pp. 341–344. ACM Press.

Henglein, F. and Mossin, Cn. (1994) Polymorphic binding-time analysis. *European Symposium on Programming*, pp. 287–301.

ICFP'97 (1997) *International Conference on Functional Programming*. ACM Press.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generations*. Prentice Hall.

Leroy, X. (1992) Unboxed objects and polymorphic typing. *Symposium on Principles of Programming Languages*, pp. 177–188.

Leroy, X. (1995) *Le système Caml Special Light: Modules et compilation efficace en Caml*. Technical report 2721, Institut National de Recherche en Informatique et Automatique.

Marlet, R., Consel, C. and Boinot, P. (1999) Efficient incremental run-time specialization for free. *Symposium on Programming Languages Design and Implementation*. ACM Press.

Milner, R., Tofte, M., Harper, B. and MacQueen, D. B. (1997) *The Definition of Standard ML revised*. MIT Press.

Minamide, Y. and Garrigue, J. (1998) On the runtime complexity of type-directed unboxing. *International Conference on Functional Programming*, pp. 1–12. ACM Press.

Moggi, E. (1988) *Computational lambda-calculus and monads*. Technical report ECS-LFCS-88-86, University of Edinburgh.

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press

Moggi, E. (1997) A categorical account of two-level languages. *13th Conference on the Mathematical Foundations of Programming Semantics.*

Moggi, E., Taha, W., Benaissa, Z. El-Abidine and Sheard, T. (1999) An idealized MetaML: simpler, and more expressive. *Proceedings European Symposium on Programming*, pp. 193–207.

Monnier, S., Blume, M. and Shao, Z. (1999) *Cross-functor inlining in FLINT.* Technical report YALEU/DCS/TR-1189, Department of Computer Science, Yale University, New Haven, CT.

Nielson, F. and Nielson, H. R. (1992) *Two-level Functional Languages.* Cambridge University Press.

Peyton Jones, S. and Marlow, S. (1999) Secrets of the Glasgow Haskell Compiler inliner. *Proceedings International Workshop on Implementation of Declarative Languages.*

PLDI'99 (1999) *Symposium on Programming Languages Design and Implementation.* ACM Press.

POPL'96 (1996) *Symposium on Principles of Programming Languages.* ACM Press.

Reynolds, J. C. (1974) Towards a theory of type structure. *Proceedings, colloque sur la programmation: Lecture Notes in Computer Science 19*, pp. 408–425. Springer-Verlag.

Saha, B. and Shao, Z. (1998) Optimal type lifting. *International Workshop on Types in Compilation.*

Shao, Z. (1997a) Flexible representation analysis. *International Conference on Functional Programming.* ACM Press.

Shao, Z. (1997b) An overview of the FLINT/ML compiler. *Proc. ACM SIGPLAN Workshop on Types in Compilation.* (Boston College Computer Science Department, Technical Report BCCS-97-03.)

Shao, Zh. (1998) Typed cross-module compilation. *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 141–152. ACM Press.

Shao, Z. (1999) Transparent modules with fully syntactic signatures. *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 220–232. ACM Press.

Shao, Z., League, C. and Monnier, S. (1998) Implementing typed intermediate languages. *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 313–323. ACM Press.

Shivers, O. (1991) *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University.

Sperber, M. and Thiemann, P. (1996) Realistic compilation by partial evaluation. *Symposium on Programming Languages Design and Implementation*, pp. 206–214. ACM Press.

Sperber, M. and Thiemann, P. (1997) Two for the price of one: Composing partial evaluation and compilation. *Symposium on Programming Languages Design and Implementation*, pp. 215–225. ACM Press.

Taha, W. (1999) *Multi-stage programming: Its theory and applications.* PhD thesis, Oregon Graduate Institute, Beaverton, Oregon.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 203–217.

Tarditi, D. (1996) *Design and implementation of code optimizations for a type-directed compiler for standard ml.* PhD thesis, Carnegie Mellon University.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. *Symposium on Principles of Programming Languages.*

https://doi.org/10.1017/S0956796802004616 Published online by Cambridge University Press