Strict Bidirectional Type Checking [*]

Adam Chlipala[†]
Computer Science Division
University of California, Berkeley
adamc@cs.berkeley.edu

Leaf Petersen[‡]
Programming Systems Laboratory
Intel Corporation
Leaf.Petersen@intel.com

Robert Harper
Computer Science Department
Carnegie Mellon University
rwh@cs.cmu.edu

Abstract

Completely annotated lambda terms (such as are arrived at via the straightforward encodings of various types from System F) contain much redundant type information. Consequently, the completely annotated forms are almost never used in practice, since partially annotated forms can be defined which still allow syntax directed type checking. An additional optimization that is used in some proof and type systems is to take advantage of the context of occurrence of terms to further elide type information using bidirectional type checking rules. While this technique is generally effective, we show that there exist bidirectional terms which exhibit asymptotic increases in the size of their type decorations when *sequentialized* into a *named-form* calculus (a common first step in compilation). In this paper, we introduce a refinement of the bidirectional type system based on *strict* logic which allows additional type decorations to be eliminated, and show that it is well-behaved under sequentialization.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Languages, Theory

Keywords

Strict Logic, Type Theory, Type Inference

[*]This research was supported in part by an NDSEG fellowship, and in part by the National Science Foundation under grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination".

[†]This work was done while the author was at Carnegie Mellon University

[‡]This work was done while the author was at Carnegie Mellon University

TLDI'05, January 10, 2005, Long Beach, California, USA.

1 Introduction

Type systems for statically typed languages are generally defined for programs which are annotated with a certain amount of type information so that type checking is a *syntax directed* process. Some source languages (such as Standard ML, OCaml, and Haskell) permit almost all type information to be elided on the external form, freeing the programmer from the task of explicitly writing type information. An explicitly annotated form of a type-free program is generally reconstructed by the compiler prior to type checking. This ability to elide type information makes the source code more concise and compact, at the expense of substantially complicating the type checking process. For more powerful type systems, full type reconstruction is not possible, and some work has been done on finding more limited forms of type reconstruction that are compatible with more powerful type systems [5].

Compactness of syntactic representation is of interest in numerous other contexts as well, such as proof systems and typed compilers. Type preserving compilers such as the TILT and FLINT Standard ML compilers make use of multiple *typed intermediate languages* in the process of compiling source files to machine code. Type information in such compilers can used for validation, certification, and optimization [4, 3, 8]. A key issue in typed compilation is controlling the size of the type information on the intermediate forms, and numerous techniques have been proposed for addressing this [6, 3, 7]. While some approaches rely on meta-level representation techniques such as hash-consing, other approaches take advantage of careful design of the object language to eliminate redundant information and to express sharing internally to the calculus. Because of the complexity of full type reconstruction (and its impossibility in more general type systems) full type reconstruction is generally not used in these languages.

In this paper we examine a type system based on strict logic which takes advantage of contextual type information while still permitting syntax-directed type checking. The most basic notion of a program considered is one in which terms are fully annotated with types wherever necessary for simple syntax-directed type checking. A well-known approach to making the syntax more compact involves using a *bidirectional type system*, in which terms are divided into *synthesis terms*, for which unique types may be determined "synthetically;" and *analysis terms*, which may be verified "analytically" to have particular types. Such type systems enjoy the simplicity of syntax-directed type checking while still successfully eliminating much unnecessary type information.

A first cut at a type system for a compiler intermediate language based on this idea causes the type annotation size benefits of a bidirectional type system to be lost. For the purposes of optimization it is important that the intermediate language maintain programs in a sequentialized form in which all intermediate computations are bound to variables [1]. This structured form is important for optimization and code generation. In the process, however, new type annotations are generated when one complex expression is broken into many simple expressions in the process of sequentialization. These new expressions are assigned to temporary variables in linear order, away from the context that allowed inference of their types in the source code.

The local inference properties provided by bidirectional type checking are lost when terms are broken apart by sequentialization. In the remainder of this paper, we suggest a solution to this problem based on *strict logic*. In order to explore these ideas with the minimum of syntactic overhead, we phrase this exposition entirely in terms of the simply typed lambda calculus. We do so with the full awareness that there are simpler techniques for dealing with the type size problem in this domain. As we discuss further in section 3.3, the intention is to use this simple setting to develop a framework which generalizes to deal with terms which do not admit simple ad-hoc solutions, and which also generalizes to incorporate higher-order types.

2 Base Language

We begin by defining the fully annotated simply typed lambda calculus [2].

$$\begin{array}{rrcl}
\text{Types:} & \tau,\sigma & ::= & b \mid \tau\to\sigma \\
\text{Variables:} & x,y,z & & \\
\text{Terms:} & e & ::= & x \mid \lambda_{\tau_1,\tau_2}x.e \mid e_1 @_{\tau_1,\tau_2} e_2 \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x:\tau
\end{array}$$

In this simple language, we can define a derived let form as $\texttt{let}\ x : \tau_2 = e_1\ \texttt{in}\ (e_2 : \tau_1)$ with $(\lambda_{\tau_2,\tau_1}x.e_2)@_{\tau_2,\tau_1}e_1$, where $\cdot \vdash e_1{:}\tau_2$ and $x:\tau_2 \vdash e_2{:}\tau_1$.

2.1 Typing Judgments

$$\frac{\Gamma(x)=\tau}{\Gamma\vdash x{:}\tau}\ \mathit{Var} \qquad \frac{\Gamma, x:\tau_1 \vdash M{:}\tau_2}{\Gamma\vdash \lambda_{\tau_1,\tau_2}x.M{:}\tau_1\to\tau_2}\ \mathit{Lambda}$$

$$\frac{\Gamma\vdash M_1{:}\tau_1\to\tau_2 \quad \Gamma\vdash M_2{:}\tau_1}{\Gamma\vdash M_1 @_{\tau_1,\tau_2} M_2{:}\tau_2}\ \mathit{App}$$

3 Bidirectional Language

It is well-known that with a slightly different formulation of language and judgments, it is possible to elide most type annotations.

We extend the language to support two different modes of typing judgment: $\Gamma\vdash e \Uparrow \tau$ means that the type τ is synthesized as "output" from the term e in context Γ. $\Gamma \vdash a \Downarrow \tau$ means that the term e may be verified to have the type τ (thought of as "input") in context Γ. This motivates the addition of some new terms and the division of terms into synthesis and analysis forms as follows:

$$\begin{array}{rrcl}
\text{Synthesis Terms:} & m_s & ::= & x \mid m_s\,(m_a) \mid [m_a{:}\tau] \\
\text{Analysis Terms:} & m_a & ::= & m_s \mid \lambda x.m_a \\
\text{Expressions:} & e & ::= & m_s \mid m_a
\end{array}$$

3.1 Typing Judgments

$$\frac{\Gamma\vdash m_s \Uparrow \tau}{\Gamma\vdash m_s \Downarrow \tau}\ \mathit{Sy} \qquad \frac{\Gamma\vdash m_a \Downarrow \tau}{\Gamma\vdash [m_a{:}\tau] \Uparrow \tau}\ \mathit{An} \qquad \frac{\Gamma(x)=\tau}{\Gamma\vdash x \Uparrow \tau}\ \mathit{Var}$$

$$\frac{\Gamma, x:\tau\vdash m_a \Downarrow \sigma}{\Gamma\vdash \lambda x.m_a \Downarrow \tau\to\sigma}\ \mathit{Lambda}$$

$$\frac{\Gamma\vdash m_s \Uparrow \tau\to\sigma \quad \Gamma\vdash m_a \Downarrow \tau}{\Gamma\vdash m_s\,(m_a) \Uparrow \sigma}\ \mathit{App}$$

The principal advantage of this type system over other systems can be seen clearly in the case of nested functions. For example, consider the standard term $\lambda_{\tau,\sigma\to\tau}x.\lambda_{\sigma,\tau}y.x$ occurring in a context such that its type is uniquely fixed by the surrounding terms. In this case, the context of occurrence defines the type of the function completely, and hence the type decorations on the functions are completely redundant. In the bidirectional system, we can elide all of this information entirely. This can be seen in the following typing derivation which shows the analysis of the function at its contextually-provided type:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma, x:\tau, y:\sigma\vdash x\Uparrow\tau}\ \mathit{Var}}{\Gamma, x:\tau, y:\sigma\vdash x\Downarrow\tau}\ \mathit{Sy}}{\Gamma, x:\tau\vdash \lambda y.x\Downarrow\sigma\to\tau}\ \mathit{Lambda}}{\Gamma\vdash\lambda x.\lambda y.x\Downarrow\tau\to\sigma\to\tau}\ \mathit{Lambda}$$

While the bidirectional system presented here is quite simple, bidirectional type checking extends easily to handle higher-order polymorphism and other advanced language features [5].

3.2 Type annotation size benefits

The bidirectional language allows us to express terms from the base language with asymptotically less type annotation. To formalize this, we define a family of functions that map terms, expressions, and types to measures of the amounts of type information they contain.

$$\begin{array}{rcl}
\mathit{Size}_{\mathsf{t}}(b) & := & 1 \\
\mathit{Size}_{\mathsf{t}}(\tau\to\sigma) & := & 1+\mathit{Size}_{\mathsf{t}}(\tau)+\mathit{Size}_{\mathsf{t}}(\sigma)
\end{array}$$

$$\begin{array}{rcl}
\mathit{Size}_{\mathsf{e}}(x) & := & 0 \\
\mathit{Size}_{\mathsf{e}}(\lambda_{\tau_1,\tau_2}x.e) & := & 1+\mathit{Size}_{\mathsf{t}}(\tau_1)+\mathit{Size}_{\mathsf{t}}(\tau_2)+\mathit{Size}_{\mathsf{e}}(e) \\
\mathit{Size}_{\mathsf{e}}(e_1 @_{\tau_1,\tau_2} e_2) & := & 1+\mathit{Size}_{\mathsf{t}}(\tau_1)+\mathit{Size}_{\mathsf{t}}(\tau_2)+\mathit{Size}_{\mathsf{e}}(e_1) \\
 & & \quad +\mathit{Size}_{\mathsf{e}}(e_2)
\end{array}$$

$$\mathit{Size}_{\mathsf{s}}(x) \ := \ 0$$

$$
\begin{array}{rcl}
\mathit{Size}_{\mathsf{s}}\left([m_a{:}\tau]\right) & := & \mathit{Size}_{\mathsf{a}}\left(m_a\right) + \mathit{Size}_{\mathsf{t}}\left(\tau\right) \\
\mathit{Size}_{\mathsf{s}}\left(m_s\left(m_a\right)\right) & := & \mathit{Size}_{\mathsf{s}}\left(m_s\right) + \mathit{Size}_{\mathsf{a}}\left(m_a\right) \\
\\
\mathit{Size}_{\mathsf{a}}\left(m_s\right) & := & \mathit{Size}_{\mathsf{s}}\left(m_s\right) \\
\mathit{Size}_{\mathsf{a}}\left(\lambda x.m_a\right) & := & \mathit{Size}_{\mathsf{a}}\left(m_a\right)
\end{array}
$$

The extra "1" terms in the cases for fully annotated function abstractions and applications treat these terms as if they had implicit arrows. This makes the definitions satisfy intuitive properties such as that $\lambda_{\tau_1,\tau_2}x.e$ and $[\lambda x.e{:}\tau_1 \to \tau_2]$ should have the same type annotation size.

In the last section, we showed a small example of a term that we can type check more efficiently with the bidirectional system using contextual information. In fact, the bidirectional language permits representations that contain asymptotically less type information than the fully annotated term. To demonstrate this, we begin by defining an indexed family of terms for which the size of the type annotation grows quadratically in the index.

Let 1 be a base type and $*$ a variable bound to a value of type 1, and let τ be some unspecified type. Define:

$$
\begin{array}{rcl}
\tau^0 & := & 1 \\
\tau^{i+1} & := & \tau \to \tau^i \\
\\
\mathsf{k}^0 & := & * \\
\mathsf{k}^{i+1} & := & \lambda_{\tau,\tau^i} x_i.\mathsf{k}^i \\
\\
\mathsf{k}_B^0 & := & * \\
\mathsf{k}_B^{i+1} & := & \lambda x_i.\mathsf{k}_B^i
\end{array}
$$

τ^i is a function type with i levels of nesting. k^i denotes a curried function that returns $*$ for any values of its i parameters of type τ. k_B^i is the bidirectional analysis term equivalent of k^i. For example, $\mathsf{k}_B^2 = \lambda x_1.\lambda x_2.*$ and $\tau^2 = \tau \to (\tau \to 1)$.

If we always analyze k_B^i at τ^i, we save significant type annotation size over k^i, with $\mathit{Size}_{\mathsf{e}}\left(\mathsf{k}^i\right) = \Theta(i^2)$ and $\mathit{Size}_{\mathsf{s}}\left([\mathsf{k}_B^i : \tau^i]\right) = \Theta(i)$. We can explain the reason for this succinctly: in the standard terms, every function parameter's type is replicated for every function abstraction up to the abstraction that binds it. It should be clear that using the bidirectional system leads to similar savings with many other terms, and that a bidirectional term need never have *more* type annotation than its fully-annotated counterpart.

3.3 A half-annotated form

The fully annotated language we use as a baseline for comparison is almost never used in practice. Thinking of the typing judgment as producing the types as outputs and taking the rest of its places as inputs, it is clear from the rules in section 2 that the only type annotation that is necessary to allow syntax-directed checking with those rules is the parameter type for a lambda abstraction. Terms of the fully annotated language can be viewed as targets of a simple type reconstruction algorithm operating on terms in other forms. This relaxation of annotations admits a particularly obvious method of regaining all of the needed types, and so it is usually taken as a starting point. Formally, it modifies the set of terms to be:

$$
\text{Terms:} \quad e \quad ::= \quad x \mid \lambda x{:}\tau.e \mid e_1\left(e_2\right)
$$

This technique of annotating terms with only their incremental contribution to the overall type of the term is also commonly used for injections into sum types, which in their fully annotated form exhibit the same bad behavior as the fully annotated lambda terms. This is not possible for all terms, however. A simple example of this can be seen with recursive functions. The half-annotation approach relies on the ability to synthesize a type from the body of the lambda: in the case that the body may contain recursive references, this synthesis is not possible without knowing the full type of the function (or without doing more complicated unification based type reconstruction). For this reason, recursive functions are generally written using a fully annotated form.

A more subtle example of this arises in the case of operations on recursive types. Consider as an example the following nested recursive type.

$$
\mu(\alpha).(\alpha+\mu(\beta).(\beta+\mu(\gamma).(\gamma+1)))
$$

Consider also a term of this type (ignoring for the time being the type annotations on the sum injections):

$$
\begin{array}{l}
\textbf{fold}[\mu(\alpha).(\alpha+\mu(\beta).(\beta+\mu(\gamma).(\gamma+1)))] \\
\quad \textbf{inr fold}[\mu(\beta).(\beta+\mu(\gamma).(\gamma+1))] \\
\quad\quad \textbf{inr fold}[\mu(\gamma).(\gamma+1)] \\
\quad\quad\quad \textbf{inr}\ *
\end{array}
$$

This example clearly exhibits exactly the same quadratic behavior seen in the previous section in the context of curried lambda terms. Unlike lambda terms, however, there is no incremental approach to marking the type annotations on fold operations that might alleviate the problem. The problem only gets worse when we add in the annotations on the sum injections.

$$
\begin{array}{l}
\textbf{fold}[\mu(\alpha).(\alpha+\mu(\beta).(\beta+\mu(\gamma).(\gamma+1)))] \\
\quad \textbf{inr}[\mu(\alpha).(\alpha+\mu(\beta).(\beta+\mu(\gamma).(\gamma+1)))] \\
\quad\quad \textbf{fold}[\mu(\beta).(\beta+\mu(\gamma).(\gamma+1))] \\
\quad\quad\quad \textbf{inr}[\mu(\beta).(\beta+\mu(\gamma).(\gamma+1))] \\
\quad\quad\quad\quad \textbf{fold}[\mu(\gamma).(\gamma+1)] \\
\quad\quad\quad\quad\quad \textbf{inr}[\mu(\gamma).(\gamma+1)]\ *
\end{array}
$$

This is the case even when (as here) we use the half-annotated version of sum injections, where the **inr** constructor is decorated only with the left half of the sum type into which it injects, relying on synthesis to produce the right half of the sum type from the subterm. Compare this term with the bidirectional version, in which the entire type need be written only once.

$$
\begin{array}{l}
[\textbf{fold inr} \\
\quad \textbf{fold inr} \\
\quad\quad \textbf{fold inr}\ * \ :\ \mu(\alpha).(\alpha+\mu(\beta).(\beta+\mu(\gamma).(\gamma+1)))]
\end{array}
$$

The advantage here is clearly substantial. While for expositional purposes we continue to use lambda terms as a running example, it should be clear that the ideas presented generalize easily to more significant problem domains in which the incremental annotation techniques used for simple lambda terms are not available.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,x:1\vdash x\Uparrow 1}}{\Gamma,x:1\vdash x\Downarrow 1}}{\Gamma\vdash\lambda x.x\Downarrow 1\to 1}}{\Gamma\vdash[\lambda x.x{:}1\to 1]\Uparrow 1\to 1}\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,x:1\vdash x\rightsquigarrow x}}{\Gamma,x:1\vdash x{:}1\rightsquigarrow x}}{\Gamma\vdash\lambda x.x{:}1\to 1\rightsquigarrow f:1\to 1=\lambda x.x;f}}{\Gamma\vdash[\lambda x.x{:}1\to 1]\rightsquigarrow f:1\to 1=\lambda x.x;f}\quad \dfrac{\dfrac{}{\Gamma\vdash *\rightsquigarrow *}}{\Gamma\vdash *{:}1\rightsquigarrow *}$$
$$\overline{\qquad\Gamma\vdash[\lambda x.x{:}1\to 1]\,(*)\rightsquigarrow f:1\to 1=\lambda x.x;f\,(*)\qquad}$$

Figure 1. An example sequentialization

4 The Sequential Language

The sequential language is a subset of the bidirectional language, with all the same types. Whereas the languages handled so far involve no mutation or other sources of effects, they easily extend to admit such possibilities. In any case, in practice a compiler generally must choose some explicit order in which to evaluate terms. The sequential form of these languages given here can be thought of as a compiler intermediate language that makes the order of term evaluation completely explicit.

$$\begin{array}{rrcl} \text{Synthesis term:} & M_s & ::= & x \mid x\,(y) \\ \text{Analysis term:} & M_a & ::= & M_s \mid \lambda x.E \\ \text{Expression:} & E & ::= & x \mid x = M_s;E \\ & & & \mid x:\tau = M_a;E \end{array}$$

The multiple bindings that are now required are identified with intermediate computations. Note that in order to type check such a term, we must have a type available with which to analyze every analysis term (since these terms rely on contextual information, rather than decorations). This is achieved here by forcing the inclusion of type annotations with bindings of terms whose types cannot be synthesized.

For example, consider the bidirectional term $[\lambda x.x{:}1\to 1]\,(*)$. An equivalent sequential term is $f:1\to 1=\lambda x.x;f\,(*)$. We see that the evaluation of the term has been decomposed into atomic operations. In this case, we can think of the operations as the allocation of a closure called f and the application of f to $*$.

4.1 Typing Judgments

$$\dfrac{\Gamma\vdash M_s\Uparrow\tau}{\Gamma\vdash M_s\Downarrow\tau}\;\textit{Sy}\qquad\dfrac{\Gamma(x)=\tau}{\Gamma\vdash x\Uparrow\tau}\;\textit{Var}$$

$$\dfrac{\Gamma,x:\tau\vdash E\Downarrow\sigma}{\Gamma\vdash\lambda x.E\Downarrow\tau\to\sigma}\;\textit{Lambda}$$

$$\dfrac{\Gamma(x)=\tau\to\sigma\quad\Gamma(y)=\tau}{\Gamma\vdash x\,(y)\Uparrow\sigma}\;\textit{App}$$

$$\dfrac{\Gamma\vdash M_s\Uparrow\sigma\quad\Gamma,x:\sigma\vdash E\Uparrow\tau}{\Gamma\vdash x=M_s;E\Uparrow\tau}\;\textit{LetS}$$

$$\dfrac{\Gamma\vdash M_a\Downarrow\sigma\quad\Gamma,x:\sigma\vdash E\Uparrow\tau}{\Gamma\vdash x:\sigma=M_a;E\Uparrow\tau}\;\textit{LetA}$$

The typing rules for this language are straightforward. As an example, the well-typedness of the example from the previous section can be derived as follows, assuming that Γ assigns the type 1 to $*$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,x:1\vdash x\Uparrow 1}\;\textit{Var}}{\Gamma,x:1\vdash x\Downarrow 1}\;\textit{Sy}}{\Gamma\vdash\lambda x.x\Downarrow 1\to 1}\;\textit{Lam}\quad\dfrac{\begin{array}{c}(\Gamma,f:1\to 1)(f)=1\to 1\\(\Gamma,f:1\to 1)(*)=1\end{array}}{\Gamma,f:1\to 1\vdash f\,(*)\Uparrow 1}\;\textit{App}}{\Gamma\vdash f:1\to 1=\lambda x.x;f\,(*)\Uparrow 1}\;\textit{LetA}$$

4.2 Sequentialization of the bidirectional language

A source term is transformed into a sequential version during compilation. Sequential representations more closely model the structure of the underlying computation and are important for optimization passes and for code generation.

We now specify how the bidirectional language may be translated to a sequential form. Translation is defined by two injections, one for translating synthesis terms and one for analysis terms. Relation $\Gamma\vdash m_s\rightsquigarrow E$ means that in context Γ synthesis term m_s translates to sequential expression E. Relation $\Gamma\vdash m_a{:}\tau\rightsquigarrow E$ has an analogous meaning, with the type τ at which to analyze the term m_a included.

Here and in other sections, we will overload the semicolon notation. We use $\overline{b}$ to denote zero or more bindings of variables, including both annotated and un-annotated bindings. $\overline{b};E$ indicates zero or more bindings followed by the expression E. In particular, it may denote E alone.

$$\dfrac{}{\Gamma\vdash x\rightsquigarrow x}\qquad\dfrac{\Gamma,x:\sigma\vdash m_a{:}\tau\rightsquigarrow E}{\Gamma\vdash\lambda x.m_a{:}\sigma\to\tau\rightsquigarrow z:\sigma\to\tau=\lambda x.E;z}$$

$$\dfrac{\Gamma\vdash m_s\Uparrow\sigma\to\tau\quad\Gamma\vdash m_s\rightsquigarrow\overline{b_1};x\quad\Gamma\vdash m_a{:}\sigma\rightsquigarrow\overline{b_2};y}{\Gamma\vdash m_s\,(m_a)\rightsquigarrow\overline{b_1};\overline{b_2};z=x\,(y)\,;z}$$

$$\dfrac{\Gamma\vdash m_a{:}\tau\rightsquigarrow E}{\Gamma\vdash[m_a{:}\tau]\rightsquigarrow E}\qquad\dfrac{\Gamma\vdash m_s\rightsquigarrow E}{\Gamma\vdash m_s{:}\tau\rightsquigarrow E}$$

Figure 1 shows a sequentialization derivation for our running example.

4.3 Terms that translate poorly to sequential form

There exist classes of bidirectional terms for which there is an asymptotically significant increase in the size of required type annotations after sequentialization. To demonstrate this, we first define type annotation size functions for the analysis terms and expressions of the sequential language. Synthesis terms contain no type annotations, so there is no need to define a size function for them.

$$
\begin{array}{rcl}
Size_{\mathsf{M}_a}(M_s) & := & 0 \\
Size_{\mathsf{M}_a}(\lambda x.E) & := & Size_{\mathsf{E}}(E) \\
\\
Size_{\mathsf{E}}(x) & := & 0 \\
Size_{\mathsf{E}}(x = M_s; E) & := & Size_{\mathsf{E}}(E) \\
Size_{\mathsf{E}}(x : \tau = M_a; E) & := & Size_{\mathsf{t}}(\tau) + Size_{\mathsf{M}_a}(M_a) + Size_{\mathsf{E}}(E)
\end{array}
$$

For analyzing source level languages, cases like the k^i family discussed earlier make bidirectional type checking a clear win. However, when these techniques are applied to sequential forms, most of the benefits disappear. Consider the sequentialization of k^i_B. Defining $y_0 = *$, we have:

$$\frac{}{\cdot \vdash \mathsf{k}^0_B{:}\tau^0 \rightsquigarrow *}$$

$$\frac{\cdot \vdash \mathsf{k}^i_B{:}\tau^i \rightsquigarrow E_i}{\cdot \vdash \mathsf{k}^{i+1}_B{:}\tau^{i+1} \rightsquigarrow y_{i+1} : \tau^{i+1} = \lambda x_{i+1}.E_i; y_{i+1}}$$

Thus, k^2_B sequentializes at τ^2 to:

$$z_1 : \tau \to (\tau \to 1) = \lambda x_1.(z_2 : \tau \to 1 = \lambda x_2.*; z_2); z_1$$

The typing information for the inner function is replicated unnecessarily.This pattern adds even more excess annotation for k^i_B with higher values of i. Even though we may analyze the sequentialization of k^i_B at τ^i, the type annotation size has returned to $\Theta(i^2)$. Inside of nested functions like k^i_B, the bidirectional type system relies on context to infer some information. In the sequential form, this nesting structure has been replaced with a linear form that more accurately reflects the structure of real computations, forfeiting the advantages nesting gave us.

5 Strict typing

The problem in the previous section arises from the fact that sequentialization separates terms from their context of use. Since the context of use provides the type at which the term is analyzed, removing the term from its context of use forces it to be annotated with a type. Note though that the *use* of the term remains unchanged under sequentialization. While the term is bound to a variable and moved to a new location, the variable is used in the same context as the original term. The bidirectional system cannot use this fact because it takes advantage only of the *local* context of occurrence of a term. The solution to the sequentialization problem lies in removing this constraint: designing a type system to take advantage of *non-local* contexts. So long as a term is bound to a variable which is used in a context which uniquely defines its type, there is no need to give a type explicitly for the term, since its type is uniquely defined by its non-local use.

This idea can be made precise using a typing formulation based on strict logic. In strict logic, as in linear logic, every hypothesis must be used: that is, the store of strict hypotheses does not admit weakening. Unlike linear logic, strict logic permits hypotheses to be used multiple times: that is, the division of the store of hypotheses among the premises of a derivation permits duplication of hypotheses.

This notion of strict logic gives rise in the natural way to a lambda calculus with the notion of a strict variable. Strict variables have the property that they may be used one or more times, but must be used at least once. Using this property, we can recover the benefits of bidirectional type checking even when term locality is lost via sequentialization.

At a high level, we do this by extending the sequential language with a $u = M_a; E$ expression form corresponding to a strict variable binding. This strict binding form omits the type annotation for the term, even though the term is an analysis term from which we are unable to synthesize a type. In this new system, the type at which to analyze this term will be determined by examining the context of occurrence of the strict variable. The strictness of the variable guarantees that there is such an occurrence, and the syntax of the language ensures that the occurrence will be in an analysis position, corresponding intuitively to the "original" locality of the term to which it is bound.

5.1 A strict lambda calculus

Standard variable:	x, y, z		
Strict variable:	u, v, w		
Variable:	q, r, s	$::=$	x, y, z, u, v, w
Strict synthesis term:	T_s	$::=$	$x \mid x(y) \mid x(u)$
Strict analysis term:	T_a	$::=$	$T_s \mid u \mid \lambda x.S$
Strict expression:	S	$::=$	$x \mid u \mid x = T_s; S$
			$\mid x : \tau = T_a; S$
			$\mid u = T_a; S$

We formalize this idea as follows. In addition to the usual typing context Γ, we include in typing judgments a strict context Δ which is a finite mapping to types from a set of strict variables. Intuitively, Δ contains variables which must be analyzed at least once in sub-judgments in order to provide a type for the term to which they are bound. In an algorithmic sense, the types in Δ are filled in using the types at which the variables are analyzed in sub-judgments and are propagated backwards to the variable binding sites.

We define $\Gamma; \Delta \vdash T_s \Uparrow \tau$ to mean that in typing context Γ and strict context Δ, the type of T_s may be synthesized as τ. $\Gamma; \Delta \vdash T_a \Downarrow \tau$ means that in typing context Γ and strict context Δ, T_a may be analyzed to have type τ. $\Delta_1 \cup \Delta_2$ denotes the mapping from a variable x to $\Delta_1(x)$ if x is present in Δ_1 or to $\Delta_2(x)$ otherwise. The two contexts joined may not map the same variable to different types.

$$\frac{\Gamma(x) = \tau}{\Gamma; \emptyset \vdash x \Uparrow \tau}\ \mathit{SVar} \qquad \frac{}{\Gamma; u : \tau \vdash u \Downarrow \tau}\ \mathit{SVarS}$$

$$\frac{\Gamma; \Delta \vdash T_s \Uparrow \tau}{\Gamma; \Delta \vdash T_s \Downarrow \tau}\ \mathit{SSy}$$

$$\frac{\Gamma, x : \sigma; \Delta \vdash S \Downarrow \tau}{\Gamma; \Delta \vdash \lambda x.S \Downarrow \sigma \to \tau}\ \mathit{SLambda}$$

$$\frac{\Gamma(x) = \sigma \to \tau \quad \Gamma(y) = \sigma}{\Gamma; \emptyset \vdash x(y) \Uparrow \tau}\ \mathit{SApp}$$

$$\frac{\Gamma(x) = \sigma \to \tau}{\Gamma; u : \sigma \vdash x(u) \Uparrow \tau}\ \mathit{SAppS}$$

$$\frac{\Gamma; \Delta_1 \vdash T_s \Uparrow \sigma \quad \Gamma, x : \sigma; \Delta_2 \vdash S \Downarrow \tau}{\Gamma; \Delta_1 \cup \Delta_2 \vdash x = T_s; S \Downarrow \tau}\ \mathit{SLetS}$$

$$\frac{\Gamma;\Delta_1 \vdash T_a \Downarrow \sigma \quad \Gamma;\Delta_2, u:\sigma \vdash S \Downarrow \tau}{\Gamma;\Delta_1 \cup \Delta_2 \vdash u = T_a; S \Downarrow \tau}\ \mathit{SLetAS}$$

$$\frac{\Gamma;\Delta_1 \vdash T_a \Downarrow \sigma \quad \Gamma, x:\sigma;\Delta_2 \vdash S \Downarrow \tau}{\Gamma;\Delta_1 \cup \Delta_2 \vdash x:\sigma = T_a; S \Downarrow \tau}\ \mathit{SLetA}$$

The key idea behind this system can be seen in the strict variable typing rule (*SVarS*). Notice that this rule requires that the strict context (Δ) contain only the variable being checked, and nothing else. Crucially, this implies that there is no implicit weakening for strict variables in this system – that is, every strict variable must be used by the *SVarS* or *SAppS* rule along at least one branch of the derivation. Since both of these rules place the strict variable in analysis positions, this corresponds exactly to the requirement that every strict variable must be used in an analysis position at least once.

The typing rules for the let constructs allow the strict context to be divided up among sub-terms as necessary via the union operation on strict contexts. The definition of strict context union (above) permits variables to be used in multiple sub-terms, but enforces the property that they must be used at the same type.

As an example, consider a typing derivation for a strict version of the term k_B^1 described earlier. Notice how the use of a strict binding saves us the use of a superfluous type annotation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x:1;\cdot \vdash * \Uparrow 1}\ \mathit{SVar}}{\Gamma, x:1;\cdot \vdash * \Downarrow 1}\ \mathit{SSy}}{\Gamma;\cdot \vdash \lambda x.* \Downarrow 1 \to 1}\ \mathit{SLam} \qquad \dfrac{}{\Gamma;\cdot, u:1\to 1 \vdash u \Downarrow 1 \to 1}\ \mathit{SVarS}}{\Gamma;\cdot \vdash u = \lambda x.*; u \Downarrow 1 \to 1}\ \mathit{SLetAS}$$

It may appear at first that this type system is not very practical. If we think of it in the same way as previous judgment systems, it seems as though an algorithm for type checking must non-deterministically “guess” a type for u, even though the point of this system is to avoid any need for non-syntax-directed inference. This is in fact not the case: there is a straightforward algorithm that we can use to type check strict terms, using only local and compositional analysis. The reason for this lies precisely in the previously discussed property of the type system: that it forces each strict variable to occur at least once in an analysis context. Formally, this enforces the property that there is a unique choice for the type at which a strict variable appears in the strict context. Intuitively, we may think of the strict context as an output of our algorithm. Type checking of a strict let binding proceeds by first checking the body of the let expression. The resulting strict context tells us the unique type at which the strict variable occurs, which can then be used to check the term to which the strict variable is bound.

5.2 An algorithm for type checking strict terms

Theorem

1. For given Γ and T_s, there is at most one Δ, τ pair such that $\Gamma;\Delta \vdash T_s \Uparrow \tau$.
2. For given Γ, T_a, and τ, there is at most one Δ such that $\Gamma;\Delta \vdash T_a \Downarrow \tau$.
3. For given Γ, S, and τ, there is at most one Δ such that $\Gamma;\Delta \vdash S \Downarrow \tau$.

Proof: By simultaneous induction on terms T_s and T_a and expressions S:

Case $T_a = T_s$: The only way to derive $\Gamma;\Delta \vdash T_s \Downarrow \tau$ is by *SSy*. By the induction hypothesis, we have that there is at most one Δ, τ pair for which this is true, so Δ is uniquely determined.

Case $T_s = x$: The only way to derive $\Gamma;\Delta \vdash x \Uparrow \tau$ is by *SVar*, which requires $\tau = \Gamma(x)$ and $\Delta = \emptyset$, uniquely determining these values as required.

Case $T_a = u$: $\Gamma;\Delta \vdash u \Downarrow \tau$ must be derived by *SVarS*, constraining Δ to be $u:\tau$.

Case $T_a = \lambda x.T_a'$: If $\Gamma;\Delta \vdash T_a \Downarrow \tau$, this must have been derived by *SLambda*, so $\tau = \tau_1 \to \tau_2$ and $\Gamma, x:\tau_1;\Delta \vdash T_a' \Downarrow \tau_2$. By the inductive hypothesis, such a judgment about T_a' is possible for at most one Δ. Since any analysis judgment of T_a requires such a judgment as a premise to a single candidate rule, we have the required result.

Case $T_s = x(y)$: If $\Gamma;\Delta \vdash x(y) \Uparrow \tau$, *SApp* must have been used. This means that Δ may only be $\emptyset$, and τ is uniquely determined by $\Gamma(x)$.

Case $T_s = x(u)$: If $\Gamma;\Delta \vdash x(u) \Uparrow \tau$, *SAppS* must have been used. This means that $\Gamma(x) = \sigma \to \tau$, uniquely determining τ, and Δ may only be $u:\sigma$.

Case $S = x = T_s; S'$: If $\Gamma;\Delta \vdash S \Downarrow \tau$, this must have been derived by *SLetS*, so $\Gamma;\Delta_1 \vdash T_s \Uparrow \sigma$ and $\Gamma, x:\sigma;\Delta_2 \vdash S' \Downarrow \tau$ for some σ, Δ_1, and Δ_2 with $\Delta = \Delta_1 \cup \Delta_2$. By the inductive hypothesis for T_s, Δ_1 and σ are uniquely determined, and thus Δ_2 is as well, by the hypothesis for S'. This means that Δ is uniquely determined. Any judgment that analyzes S to have type τ must use *SLetS* with premises of the sorts described above, so we have the required result.

Case $S = x:\sigma = T_a'; S'$: If $\Gamma;\Delta \vdash S \Downarrow \tau$, this must have been derived by *SLetA*, so $\Gamma;\Delta_1 \vdash T_a' \Downarrow \sigma$ and $\Gamma, x:\sigma;\Delta_2 \vdash S' \Downarrow \tau$ for some Δ_1 and Δ_2 with $\Delta = \Delta_1 \cup \Delta_2$. By the inductive hypotheses for T_a' and S', Δ_1 and Δ_2 are uniquely determined. This means that Δ is uniquely determined. Any judgment that analyzes S to have type τ must use *SLetA* with premises of the sorts described above, so we have the required result.

Case $S = u = T_a'; S'$: If $\Gamma;\Delta \vdash S \Downarrow \tau$, this must have been derived by *SLetAS*, so $\Gamma;\Delta_1 \vdash T_a' \Downarrow \sigma$ and $\Gamma;\Delta_2, u:\sigma \vdash S' \Downarrow \tau$ for some σ, Δ_1, and Δ_2 with $\Delta = \Delta_1 \cup \Delta_2$. By the inductive hypothesis for S', Δ_2 and σ are uniquely determined, and thus Δ_1 is as well, by the hypothesis for T_a'. This means that Δ is uniquely determined. Any judgment that analyzes S to have type τ must use *SLetAS* with premises of the sorts described above, so we have the required result. □

Thus, the strict typing rules describe a deterministic algorithm for type checking strict terms. The elements of judgments shown to be uniquely determined are outputs and the rest inputs. The order in which inductive hypotheses are applied above suggests the order for checking sub-terms with recursive applications of the algorithm. As an optimization, the unions of strict contexts found above could be replaced by incremental modification of a strict context, taking it as an input to the algorithm and outputting a strengthened version.

5.3 Sequentialization

Now we may define a sequentialization process that produces strict terms. Where $\overline{b}$ is a prefix of a strict expression, x is a regular variable, τ a type, and u a strict variable, $\overline{b}[x:\tau/u]$ denotes $\overline{b}$ with the binding of u changed to bind x with annotation τ and with later occurrences of u changed to occurrences of x.

The key insight is this: Since we restricted function abstractions to be analysis terms in the original bidirectional language, we know that context will always determine their types. Therefore, we can bind each intermediate function value to a strict variable, eliminating the need for annotations.

$$\frac{}{\Gamma \vdash x \mapsto x}$$

$$\frac{\Gamma \vdash m_s \Uparrow \sigma \to \tau \quad \Gamma \vdash m_s \mapsto \overline{b_1};y \quad \Gamma \vdash m_a{:}\sigma \mapsto \overline{b_2};r}{\Gamma \vdash m_s\,(m_a) \mapsto \overline{b_1};\overline{b_2};x = y(r);x}$$

$$\frac{\Gamma \vdash m_s \mapsto S}{\Gamma \vdash m_s{:}\tau \mapsto S} \qquad \frac{\Gamma, x:\sigma \vdash m_a{:}\tau \mapsto S}{\Gamma \vdash \lambda x.m_a{:}\sigma \to \tau \mapsto u = (\lambda x.S);u}$$

$$\frac{\Gamma \vdash m_a{:}\tau \mapsto \overline{b};x}{\Gamma \vdash [m_a{:}\tau] \mapsto \overline{b};x} \qquad \frac{\Gamma \vdash m_a{:}\tau \mapsto \overline{b};u}{\Gamma \vdash [m_a{:}\tau] \mapsto \overline{b}[x:\tau/u];x}$$

It is worth mentioning why the application rule is complete. It requires that a particular synthesis term linearizes to an expression ending in a standard variable. Though we don't prove it here, it's not hard to see that any synthesis term sequentializes in such a way.

5.4 Translation size benefits

We can now revisit the earlier example of a class of bidirectional terms with an asymptotically bad type annotation size blowup in sequentialization. We show how our running example is better handled with this new sequentialization to give an intuitive idea of its benefits, and we prove a general theorem in the next section.

First, we define a reasonable measure of the size of type annotations on strict analysis terms and expressions. The strict language is formulated so that these are the only sizes worth defining, since synthesis terms contain no type annotations.

$$\begin{array}{rcl} Size_{\mathsf{T}_a}(\lambda x.S) & := & Size_{\mathsf{S}}(S) \\ Size_{\mathsf{T}_a}(T_s) & := & 0 \\ \\ Size_{\mathsf{S}}(x = T_s;S) & := & Size_{\mathsf{S}}(S) \\ Size_{\mathsf{S}}(u = T_a;S) & := & Size_{\mathsf{T}_a}(T_a) + Size_{\mathsf{S}}(S) \\ Size_{\mathsf{S}}(x:\tau = T_a;S) & := & Size_{\mathsf{T}_a}(T_a) + Size_{\mathsf{t}}(\tau) \\ & & \quad + Size_{\mathsf{S}}(S) \end{array}$$

With the strict translation, we have:

$$\frac{}{\cdot \vdash \mathsf{k}_B^0{:}\tau^0 \mapsto *} \qquad \frac{\cdot \vdash \mathsf{k}_B^{i-1}{:}\tau^{i-1} \mapsto S_{i-1}}{\cdot \vdash \mathsf{k}_B^i{:}\tau^i \mapsto u_i = (\lambda x_i.S_{i-1});u_i}$$

It is easily verified that $Size_{\mathsf{T}_a}(S^i) = Size_{\mathsf{a}}(k_B^i)$, with $\cdot \vdash \mathsf{k}_B^i{:}\tau^i \mapsto S^i$ for every i, in contrast with the asymptotically significant annotation increase with the earlier sequentialization.

5.5 A bound on type annotation size of a sequentialization

It is now possible to prove that the benefit observed in the above case holds for all terms. In particular, sequentializing a base bidirectional term into a strict term will never lead to an increase in type annotation size.

Combination Lemma: For strict expression prefixes $\overline{b_1}$ and $\overline{b_2}$, variables q and r, and strict expression S, $Size_{\mathsf{S}}(\overline{b_1};\overline{b_2};S) = Size_{\mathsf{S}}(\overline{b_1};q) + Size_{\mathsf{S}}(\overline{b_2};r) + Size_{\mathsf{S}}(S)$.

Proof: By a straightforward induction over the lengths of $\overline{b_1}$ and $\overline{b_2}$. $\square$

Theorem:

1. For given Γ and m_s, if $\Gamma \vdash m_s \mapsto S$, then
$$Size_{\mathsf{S}}(S) \le Size_{\mathsf{s}}(m_s)$$
2. For given Γ, m_a, and τ, if $\Gamma \vdash m_a{:}\tau \mapsto S$, then
$$Size_{\mathsf{S}}(S) \le Size_{\mathsf{a}}(m_a)$$

Proof: By induction on the structure of terms m_s and m_a:

Case $m_a = m_s$: For given Γ, if $\Gamma \vdash m_s{:}\tau \mapsto S$, then it must be that $\Gamma \vdash m_s \mapsto S$. The induction hypothesis yields the desired result.

Case $m_s = x$: For given Γ, we have $\Gamma \vdash x \mapsto x$. $Size_{\mathsf{S}}(x) = 0 \le 0 = Size_{\mathsf{s}}(x)$.

Case $m_a = m_s\,(m'_a)$: If $\Gamma \vdash m_a \mapsto S$, it must be that $\Gamma \vdash m_s \Uparrow \sigma \to \tau$, $\Gamma \vdash m_s \mapsto \overline{b_1};q$, and $\Gamma \vdash m'_a{:}\sigma \mapsto \overline{b_2};r$, with $S = \overline{b_1};\overline{b_2};x = q(r);x$. By the Combination Lemma:

$$\begin{array}{l} Size_{\mathsf{S}}(S) \\ \quad = Size_{\mathsf{S}}(\overline{b_1};q) + Size_{\mathsf{S}}(\overline{b_2};r) + Size_{\mathsf{S}}(x = q(r);x) \\ \quad = Size_{\mathsf{S}}(\overline{b_1};q) + Size_{\mathsf{S}}(\overline{b_2};r) \end{array}$$

By hypothesis,

$$Size_{\mathsf{S}}(\overline{b_1};q) \le Size_{\mathsf{s}}(m_s)$$

and

$$Size_{\mathsf{S}}(\overline{b_2};r) \le Size_{\mathsf{a}}(m'_a)$$

By definition,

$$Size_{\mathsf{s}}(m_a) = Size_{\mathsf{s}}(m_s) + Size_{\mathsf{a}}(m'_a)$$

Therefore,

$$\begin{aligned}&Size_{\mathsf{S}}(S)\\&\quad = Size_{\mathsf{S}}\left(\overline{b_1};q\right) + Size_{\mathsf{S}}\left(\overline{b_2};r\right)\\&\quad \leq Size_{\mathsf{s}}(m_s) + Size_{\mathsf{a}}(m'_a) = Size_{\mathsf{s}}(m_a)\end{aligned}$$

Which is the needed result.

Case $m_a = \lambda x.m'_a$**:** If $\Gamma \vdash m_a{:}\sigma \to \tau \mapsto S$, it must be that $\Gamma, x : \sigma \vdash m'_a{:}\tau \mapsto S'$ and $S = u = (\lambda x.S');u$. By definition, $Size_{\mathsf{S}}(S) = Size_{\mathsf{S}}(S')$ and $Size_{\mathsf{a}}(m_a) = Size_{\mathsf{a}}(m'_a)$. By hypothesis, $Size_{\mathsf{S}}(S') \leq Size_{\mathsf{a}}(m'_a)$. Thus, $Size_{\mathsf{S}}(S) \leq Size_{\mathsf{a}}(m_a)$.

Case $m_a = [m'_a{:}\tau]$**:** If $\Gamma \vdash m_a \mapsto S$, it must be that $\Gamma \vdash m'_a{:}\tau \mapsto S'$, where $S' = \overline{b};q$.

If q is some normal variable x, then $S = \overline{b};x$. This makes $Size_{\mathsf{S}}(S) = Size_{\mathsf{S}}(S')$. By hypothesis, $Size_{\mathsf{S}}(S') \leq Size_{\mathsf{a}}(m'_a)$. Therefore, since $Size_{\mathsf{a}}(m_a) = Size_{\mathsf{a}}(m'_a) + Size_{\mathsf{t}}(\tau)$, we have the needed $Size_{\mathsf{S}}(S) \leq Size_{\mathsf{s}}(m_a)$.

Otherwise, q is some strict variable u, and $S = \overline{b}[x : \tau/u];x$ for a new x. This means $Size_{\mathsf{S}}(S) = Size_{\mathsf{S}}(S') + Size_{\mathsf{t}}(\tau)$. Using the hypothesis $Size_{\mathsf{S}}(S') \leq Size_{\mathsf{a}}(m'_a)$ again with $Size_{\mathsf{a}}(m_a) = Size_{\mathsf{a}}(m'_a) + Size_{\mathsf{t}}(\tau)$, we have the needed $Size_{\mathsf{S}}(S) \leq Size_{\mathsf{s}}(m_a)$. $\square$

6 Conclusion

We have shown that there exist transformations on programs under which the folklore technique of bidirectional type checking behaves poorly, and shown that with some additional mechanism it is possible to recover the original asymptotic behavior in a relatively simple type system requiring no unification. The strict type system and language presented here provide a compact format for expressing intermediate forms of source level programs. It is straightforward to translate terms in a language with a standard bidirectional type system into strict terms without inflating the amount of space used to store type annotations.

In addition to the function types used as an expository example here, the strict bidirectional system extends easily to handle polymorphism, dependent function types, recursive types, recursive functions, and any other construct that can be handled with a normal bidirectional type system [5]. The reason for this extensibility can be understood by observing that bidirectional type checking simply provides a framework for eliminating redundant information in terms, without regard to the particular nature of the terms or their types. The strict variant of the bidirectional system then provides a mechanism for using information from non-local contexts within the bidirectional framework.

The original motivation for the study of this type system arose out of the need for more compact representations in the TILT internal language, and we believe that the use of strict typing can be useful in this and other similar contexts. We also believe that it may be interesting to consider uses of strict typing in language design. Pierce and Turner [5] studied a number of such partial type reconstruction techniques, including the bidirectional system that provides our starting point. Strict bidirectional typing extends this work naturally to allow bidirectional local type inference to take advantage of non-local uses of term bindings.

7 References

[1] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247. ACM Press, 1993.

[2] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

[3] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[4] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[5] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, 1998.

[6] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[7] Christopher A. Stone and Robert Harper. Deciding Type Equivalence in a Language with Singleton Kinds. Technical Report CMU-CS-99-155, Department of Computer Science, Carnegie Mellon University, 1999.

[8] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.